

Preface

This volume contains the papers presented at HotSpot 2016: Hot Issues in Security Principles and Trust held on April 4-7, 2016 in Eindhoven. This workshop is intended to be a less formal counterpart to the Principles of Security and Trust (POST) conference at ETAPS, and with an emphasis on "hot topics", both of security and of its theoretical foundations and analysis.

Like POST, the themes are:

- theory of computer security;
- formal specification, analysis and design of security systems;
- automated reasoning for security analysis.

Submissions about new and emerging topics (for example, those that have not appeared prominently in conferences and workshops until now) are particularly encouraged. The PC members selected 8 papers out of the submitted papers and we hope you will enjoy the program!

Note: the management of the submissions as well as the proceedings have been done thanks to the Easychair platform.

March 8, 2016
Nancy, France

Veronique Cortier

Table of Contents

A Method for Verifying Privacy-Type Properties: The Unbounded Case . . .	1
<i>Lucca Hirschi, David Baelde and Stéphanie Delaune</i>	
Learning and Verifying Unwanted Behaviours	17
<i>Wei Chen, David Aspinall, Andrew D. Gordon, Charles Sutton and Igor Muttik</i>	
Towards Safe Enclaves	33
<i>Neline van Ginkel, Raoul Strackx, Jan Tobias Muehlberg and Frank Piessens</i>	
Verification of privacy-type properties for security protocols with XOR . .	49
<i>David Baelde, Stéphanie Delaune, Ivan Gazeau and Steve Kremer</i>	
Principles of Layered Attestation	52
<i>Paul Rowe</i>	
Measuring Protocol Strength with Security Goals	74
<i>Paul Rowe, Joshua Guttman and Moses Liskov</i>	
Localizing Security for Distributed Firewalls	97
<i>Pedro Adão, Riccardo Focardi, Joshua Guttman and Flaminia Luccio</i>	
Securing IoT communications: at what cost?	108
<i>Chiara Bodei and Letterio Galletta</i>	

Program Committee

Hubert Comon-Lundh	ENS Cachan
Veronique Cortier	CNRS, Loria
Cas Cremers	University of Oxford
Pierpaolo Degano	Dipartimento di Informatica - Universita' di Pisa
Riccardo Focardi	Università Ca' Foscari, Venezia
Sibylle Froeschle	University of Oldenburg
Joshua Guttman	Worcester Polytechnic Institute
Ralf Kuesters	University of Trier
Boris Köpf	IMDEA Software Institute
Catherine Meadows	NRL
Charles Morisset	Newcastle University
P. Y. A. Ryan	University of Luxembourg
Pierangela Samarati	Università degli Studi di Milano
Steve Schneider	University of Surrey
Geoffrey Smith	Florida International University

Additional Reviewers

Fett, Daniel
Foresti, Sara
Moran, Murat
Pelosi, Gerardo
Schmitz, Guido

Author Index

Adão, Pedro	97
Aspinall, David	17
Baelde, David	1, 49
Bodei, Chiara	108
Chen, Wei	17
Delaune, Stéphanie	1, 49
Focardi, Riccardo	97
Galletta, Letterio	108
Gazeau, Ivan	49
Gordon, Andrew D.	17
Guttman, Joshua	74, 97
Hirschi, Lucca	1
Kremer, Steve	49
Liskov, Moses	74
Luccio, Flaminia	97
Muehlberg, Jan Tobias	33
Muttik, Igor	17
Piessens, Frank	33
Rowe, Paul	52, 74
Strackx, Raoul	33
Sutton, Charles	17
van Ginkel, Neline	33

A Method for Verifying Privacy-Type Properties: The Unbounded Case*

Lucca Hirschi, David Baelde, and Stéphanie Delaune

LSV, ENS Cachan & CNRS

{hirschi,baelde,delaune}@lsv.ens-cachan.fr

Abstract

In this paper, we consider the problem of verifying anonymity and unlinkability in the symbolic model for an unbounded number of sessions, where protocols are represented as processes in a variant of the applied pi calculus, notably used in the ProVerif tool. Existing tools and techniques do not allow to verify directly these properties, expressed as behavioral equivalences. We propose a different approach: we design two conditions on protocols which are sufficient to ensure anonymity and unlinkability, and which can then be effectively checked automatically using *e.g.* ProVerif. Our two conditions correspond to two broad classes of attacks on unlinkability, *i.e.* data and control-flow leaks. This theoretical result is general enough that it applies to a wide class of protocols. In particular, we apply our techniques to provide first-time formal security proofs of protocols such as BAC (e-passport), Hash-Lock (RFID authentication), etc. Our work has also lead to the discovery of new attacks, including one on the LAK protocol (RFID authentication) which was previously claimed to be unlinkable (in a weak sense).

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases security protocols, formal methods, symbolic model, privacy, unlinkability, anonymity

1 Introduction

Security protocols aim at securing communications over various types of insecure networks (*e.g.* web, wireless devices, etc.) where dishonest users may listen to communications and interfere with them. A *secure communication* has a different meaning depending on the underlying application. It ranges from the confidentiality of data (medical files, secret keys, etc.) to *e.g.* verifiability in electronic voting systems. Another example of a security notion is privacy. In this paper, we focus on two privacy-related properties, namely unlinkability (sometimes called untraceability), and anonymity. These two notions are informally defined in the ISO/IEC standard 15408 [1] as follows:

- Unlinkability aims at *ensuring that a user may make multiple uses of a service or resource without others being able to link these uses together.*
- Anonymity aims at *ensuring that a user may use a service or resource without disclosing its identity.*

Both are critical for instance for Radio-Frequency Identification Devices (RFID) and are notably extensively studied in that context (see *e.g.* [30] for a survey of attacks on this type of protocols) but they obviously are not limited to it.

One extremely successful approach when designing and analyzing security protocols, is the use of formal methods. The purpose of formal verification is to provide rigorous frameworks and techniques to analyze protocols. For example, a flaw has been discovered in

* Extended abstract, full paper at [29].



the Single-Sign-On protocol used *e.g.* by Google Apps. It has been shown that a malicious application could very easily access to any other application (*e.g.* Gmail or Google Calendar) of their users [5]. This flaw has been found when analyzing the protocol using formal methods, abstracting messages by a term algebra and using the Avantssar validation platform. Another example is a flaw on vote-privacy discovered during the formal and manual analysis of an electronic voting protocol [18]. All these results have been obtained using *formal symbolic models*, where most of the cryptographic details are ignored using abstract structures. The techniques used in symbolic models have become mature and several tools for protocol verification are nowadays available, *e.g.* the Avantssar platform [6], the Tamarin prover [25], and the ProVerif tool [11].

Unfortunately, most of these results and tools focus on trace properties, that is, statements that something bad never occurs on any execution trace of a protocol. Secrecy and authentication are typical examples of trace properties: a data remains confidential if, for any execution, the attacker is not able to produce the data. But privacy properties like unlinkability and anonymity are not defined as trace properties. These properties are usually defined as the fact that an observer cannot distinguish between two situations, and requires a notion of behavioral equivalence. Roughly, two protocols P and Q are equivalent if an attacker cannot observe any difference between P and Q . Based on such a notion of equivalence, several definitions of privacy-type properties have been proposed (*e.g.* [4, 14] for unlinkability, *e.g.* [20, 7] for vote-privacy). In this paper, we consider the definitions of strong unlinkability and anonymity as defined in [4].

Considering an unbounded number of sessions, the problem of deciding whether a protocol satisfies an equivalence property is undecidable even for a very limited fragment of protocols (see *e.g.* [17]). Bounding the number of sessions suffices to retrieve decidability for standard primitives (see *e.g.* [9, 16]). However, analyzing a protocol for a fixed (often low) number of sessions does not allow to prove security. Moreover, in case of equivalence properties, the existing tools scale badly and can only analyze protocols for a very limited number of sessions, typically 2 or 3. Another approach consists in implementing a procedure that is not guaranteed to terminate. This is in particular the case of ProVerif, a well-established tool for checking security of protocols. ProVerif is able to check a strong notion of equivalence (called diff-equivalence) between processes that have the same structure. ProVerif has been recently extended [15] to conclude more often. Despite this recent effort intended to prove unlinkability of the BAC protocol (used in e-passport), ProVerif can still not be used off-the-shelf to establish unlinkability properties, and therefore cannot conclude on the case studies given in Section 4. Recently, similar approaches have been implemented in two other tools, namely Tamarin [8] and Maude-NPA [28]. They are based on a notion of diff-equivalence, and therefore suffer from the same drawbacks.

In this paper, we follow a different approach. We aim at proposing sufficient conditions that can be automatically checked, and that imply unlinkability and anonymity of the protocol under study. This approach is in the same spirit as the one presented in [14]. However, the class of protocols we target is quite different. For instance, in [14], they are only able to consider a very restricted class of protocols (single-step protocols that only use hash function as cryptographic primitives). We target more complex protocols and the success of our solution will be measured by confronting it to many case studies.

Our contribution

We identify a large class of 2-party protocols (protocols with else branches, arbitrary cryptographic primitives) and we devise two conditions that imply unlinkability and anonymity for

an unbounded number of sessions. We show how these two conditions can be automatically checked using the ProVerif tool, and we provide tool support for that. We have analyzed several protocols, among them the Basic Access Control (BAC) protocol as well as the Password Authenticated Connection Establishment (PACE) protocol used in the e-passport application. It happens that our conditions are rather tight and each time one of our condition was not satisfied, we report on an attack. We notably establish the first proof of unlinkability for the BAC protocol followed by the Passive Authentication (PA) and Active Authentication (AA) protocols. We also report on an attack that we found on the LAK protocol whereas it is claimed untraceable in [30] and attacks on some versions of PACE.

Let us now give an intuitive overview of our conditions. In order to do this, assume that we want to design a mutual authentication protocol between a tag T and a reader R based on symmetric encryption, and we want this protocol to be unlinkable. We note $\{m\}_k$ the symmetric encryption of a message m with a key k and we assume that k is a symmetric key shared between T and R .

A first attempt to design such a protocol is presented using Alice & Bob notation as follows (n_T is a fresh nonce):

1. $R \rightarrow T: n_R$
2. $T \rightarrow R: \{n_R\}_k$

This first attempt based on a challenge-response scheme is actually linkable. Indeed, an active attacker who systematically intercepts the nonce n_R and replaces it by a constant will be able to infer whether the same tag has been used in different sessions or not by comparing the answers he receives. Here, the tag is linkable because, for a certain behavior (possibly malicious) of the attacker, some relations between messages leak information about the agents that are involved in the execution. Our first condition, namely *frame opacity*, actually checks that all outputted messages have only trivial relations that can therefore not be exploited by the attacker.

Our second attempt takes the previous attack into account and randomizes the tag's response (using a nonce n_T) and should achieve mutual authentication by requiring that the reader must answer to the challenge n_T . This protocol can be as follows:

1. $R \rightarrow T: n_R$
2. $T \rightarrow R: \{n_R, n_T\}_k$
3. $R \rightarrow T: \{n_T\}_k$
4. $T \rightarrow R: \dots$

Here, Alice & Bob notation shows its limit. It does not specify how the reader and the tag are supposed to check that the messages they received are of the expected form, and how they should react when the messages are not well formed. This has to be precisely defined, since unlinkability depends on it. For instance, assume the tag does not check that the message he receives at step 3 contains n_T , and aborts the session if the received message is not encrypted with its own k . In such an implementation, an active attacker can eavesdrop a message $\{n_T\}_k$ sent by R to a tag T , and try to inject this message at the third step of another session played by T' . The tag T' will react by either aborting or by continuing the execution of this protocol. Depending on the reaction of the tag, the attacker will be able to infer if T and T' are the same tag or not.

In this example, the attacker adopts a malicious behavior that is not detected immediately by the tag who keeps executing the protocol. The fact that the tag passes successfully a conditional reveals crucial information about the agents that are involved in the execution.

Our second condition, namely *well-authentication*, basically requires that when an execution deviates from the honest one, the agents that are involved cannot successfully pass a conditional.

Our main theorem states that these two conditions, frame opacity and well-authentication, are actually sufficient to ensure both unlinkability and anonymity. This theorem is of interest as our conditions are fundamentally simpler (frame opacity can now be expressed using diff-equivalence and well-authentication is a trace property) and are now in the scope of existing automatic verification tools like ProVerif.

Outline

In Section 2, we briefly present our model inspired from the applied pi calculus as well as the notion of trace equivalence. We then define in Section 3 the class of protocols and the formal definitions of unlinkability and anonymity we study in this paper. Our two conditions (frame opacity and well-authentication) and our main theorem are given in Section 4. We also summarize results on our case studies. Finally, we conclude in Section 5.

2 Model

We shall model security protocols using a process algebra inspired from the applied pi calculus [2]. More specifically, we consider the calculus of Blanchet *et al.* [13], which is used in the ProVerif tool. Participants in a protocol are modeled as processes, and the communication between them is modeled by means of the exchange of messages that are represented by a term algebra.

2.1 Term algebra

We now present term algebras, which will be used to model messages built and manipulated using various cryptographic primitives. We consider an infinite set \mathcal{N} of *names* which are used to represent keys, and nonces; and two infinite and disjoint sets of *variables*, denoted \mathcal{X} and \mathcal{W} . Variables in \mathcal{X} will typically be used to refer to unknown parts of messages expected by participants, while variables in \mathcal{W} will be used to store messages learned by the attacker. We assume a *signature* Σ , *i.e.* a set of function symbols together with their arity. The elements of Σ are split into *constructor* and *destructor* symbols, *i.e.* $\Sigma = \Sigma_c \sqcup \Sigma_d$.

Given a signature \mathcal{F} , and a set of initial data A , we denote by $\mathcal{T}(\mathcal{F}, A)$ the set of terms built from elements of A by applying function symbols in \mathcal{F} . Terms of $\mathcal{T}(\Sigma_c, \mathcal{N} \cup \mathcal{X})$ will be called *constructor terms*. We denote $\text{vars}(u)$ the set of variables that occur in a term u . A *message* is a constructor term u that is *ground*, *i.e.* such that $\text{vars}(u) = \emptyset$. We denote by \bar{x} , \bar{n} , \bar{u} a (possibly empty) sequence of variables, names, and terms respectively. The application of a substitution σ to a term u is written $u\sigma$, and we denote $\text{dom}(\sigma)$ its *domain*. The *positions* of a term are defined as usual.

► **Example 1.** Consider the signature

$$\Sigma = \{\text{enc}, \text{dec}, \langle \rangle, \pi_1, \pi_2, \oplus, 0, \text{eq}, \text{ok}\}.$$

The symbols **enc** and **dec** of arity 2 represent symmetric encryption and decryption. Pairing is modeled using $\langle \rangle$ of arity 2, whereas projection functions are denoted π_1 and π_2 , both of arity 1. The function symbol \oplus of arity 2 and the constant 0 are used to model the exclusive or operator. Finally, we consider the symbol **eq** of arity 2 to model equality test, as well as the constant symbol **ok**. This signature is split into two parts: $\Sigma_c = \{\text{enc}, \langle \rangle, \oplus, 0, \text{ok}\}$, and $\Sigma_d = \{\text{dec}, \pi_1, \pi_2, \text{eq}\}$.

As in the process calculus presented in [13], constructor terms are subject to an equational theory; this has proved very useful for modeling algebraic properties of cryptographic primitives (see *e.g.* [19] for a survey). Formally, we consider a congruence $=_E$ on $\mathcal{T}(\Sigma_c, \mathcal{N} \cup \mathcal{X})$, generated from a set of equations E over $\mathcal{T}(\Sigma_c, \mathcal{X})$. Thus, $=_E$ is closed under substitutions and under bijective renaming. We finally assume that there exist u, v such that $u \neq_E v$.

► **Example 2.** To reflect the algebraic properties of the exclusive or operator, we may consider the equational theory generated by the following equations:

$$\begin{array}{ll} x \oplus 0 &= x & (x \oplus y) \oplus z &= x \oplus (y \oplus z) \\ x \oplus x &= 0 & (x \oplus y) &= (y \oplus x) \end{array}$$

In such a case, we have that $\text{enc}(a \oplus (b \oplus a), k) =_E \text{enc}(b, k)$.

We may also want to give a meaning to destructor symbols. For this, we consider the notion of *computation relation*. This relations usually enjoy nice properties but its details are unimportant for this paper.

► **Definition 3.** A *computation relation* is a relation over $\mathcal{T}(\Sigma, \mathcal{N}) \times \mathcal{T}(\Sigma_c, \mathcal{N})$, denoted \Downarrow .

The relation \Downarrow associates, to any ground term t , at most one message up to the equational theory E . For a ground term t , when there is no message m such that $t \Downarrow m$, we say that the *computation fails*; this is denoted $t \Downarrow$.

A computation relation is often obtained from a *rewriting system*, *i.e.* a set of rewriting rules of the form $g(u_1, \dots, u_n) \rightarrow u$ where g is a destructor, and $u, u_1, \dots, u_n \in \mathcal{T}(\Sigma_c, \mathcal{X})$. A ground term t can be rewritten into t' if there is a position p in t and a rewriting rule $g(u_1, \dots, u_n) \rightarrow u$ such that $t|_p = g(v_1, \dots, v_n)$ and $v_1 =_E u_1\theta, \dots, v_n =_E u_n\theta$ for some substitution θ , and $t' = t[u\theta]_p$ (*i.e.* t in which the sub-term at position p has been replaced by $u\theta$). Moreover, we assume that $u_1\theta, \dots, u_n\theta$ as well as $u\theta$ are messages.

► **Example 4.** Continuing Example 1, the properties of symbols in Σ_d are reflected through the following rewriting rules:

$$\begin{array}{ll} \text{dec}(\text{enc}(x, y), y) &\rightarrow x & \text{eq}(x, x) &\rightarrow \text{ok} \\ \pi_i(\langle x_1, x_2 \rangle) &\rightarrow x_i & \text{for } i \in \{1, 2\}. \end{array}$$

This rewriting system is convergent modulo the equational theory E given in Example 2, and induces a computation relation. For instance, we have that $\text{dec}(\text{enc}(c, a \oplus b), b \oplus a) \Downarrow c$, whereas $\text{dec}(\text{enc}(c, a \oplus b), b) \Downarrow$, and $\text{dec}(a, b) \oplus \text{dec}(a, b) \Downarrow$.

Our notion of computation relation is enough generic to deal with a destructor symbol neq , and consider that $\text{neq}(u, v) \Downarrow \text{ok}$ if, and only if, u and v can be reduced to messages that are not equal modulo E .

For modeling purposes, we split the signature Σ into two parts, namely Σ_{pub} and Σ_{priv} . An attacker builds his own messages by applying public function symbols to terms he already knows and that are available through variables in \mathcal{W} . Formally, a computation done by the attacker is a *recipe*, *i.e.* a term in $\mathcal{T}(\Sigma_{\text{pub}}, \mathcal{W})$. Recipes will be denoted by R, M, N . Note that, although we do not give the attacker the ability to generate fresh names to use in recipes, we obtain essentially the same capability by assuming an infinite supply of public constants in $\Sigma_c \cap \Sigma_{\text{pub}}$.

2.2 Process algebra

We consider a set \mathcal{C} of channel names that are assumed to be public. Protocols are modeled through processes using the grammar in Figure 1.

P, Q	$:=$	0	null
		$\text{in}(c, x).P$	input
		$\text{out}(c, u).P$	output
		$\text{let } \bar{x} = \bar{v} \text{ in } P \text{ else } Q$	evaluation
		$P \mid Q$	parallel
		$!P$	replication
		$\nu n.P$	restriction

where $c \in \mathcal{C}$, $x \in \mathcal{X}$, $n \in \mathcal{N}$, $u \in \mathcal{T}(\Sigma_c, \mathcal{N} \cup \mathcal{X})$ is a constructor term, \bar{x} (resp. \bar{v}) is a sequence of variables in \mathcal{X} (resp. terms in $\mathcal{T}(\Sigma, \mathcal{N} \cup \mathcal{X})$) both of the same length.

■ **Figure 1** Syntax of processes

Most of the constructions are rather standard. We may note the special construct $\text{let } \bar{x} = \bar{v} \text{ in } P \text{ else } Q$ that combines several standard constructions, allowing to write computations and conditionals compactly. Such a process tries to evaluate the sequence of terms \bar{v} and in case of success, *i.e.* when $\bar{v} \Downarrow \bar{u}$ for some messages \bar{u} , the process P in which \bar{x} are replaced by \bar{u} is executed; otherwise the process Q is executed. The goal of this construct is to avoid nested let instructions to be able to define our class of protocols in a simple way later on. Note also that the let instruction together with the **eq** theory as defined in Example 4 can encode the usual conditional construction. Indeed, “ $\text{let } x = \text{eq}(u, v) \text{ in } P \text{ else } Q$ ” will execute P only if the computation succeeds on $\text{eq}(u, v)$, that is only if $u \Downarrow u'$, $v \Downarrow v'$, and $u' =_{\text{E}} v'$ for some messages u' and v' .

For brevity, we sometimes omit “else 0” and null processes after outputs. We write $fv(P)$ for the set of *free variables* of P , *i.e.* the set of variables that are not in the scope of an input or a let construct. A process P is ground if $fv(P) = \emptyset$.

► **Example 5.** We consider the RFID protocol due to Feldhofer *et al.* as described in [22] and which can be presented using Alice & Bob notation as follows:

1. $I \rightarrow R: n_I$
2. $R \rightarrow I: \{n_I, n_R\}_k$
3. $I \rightarrow R: \{n_R, n_I\}_k$

The protocol is between an initiator I (the reader) and a responder R (the tag) that share a symmetric key k . We consider the term algebra introduced in Example 1 (with a trivial relation \Downarrow equals to the identity over messages). The protocol is modeled by the parallel composition of the processes P_I and P_R , corresponding respectively to the roles I and R .

$$P_{\text{Fh}} := \nu k. (\nu n_I. P_I \mid \nu n_R. P_R)$$

where P_I and P_R are defined as follows, with $u = \text{dec}(x_1, k)$:

$$\begin{aligned} P_I &:= \text{out}(c_I, n_I). \text{in}(c_I, x_1). \\ &\quad \text{let } x_2, x_3 = \text{eq}(n_I, \pi_1(u)), \pi_2(u) \text{ in} \\ &\quad \text{out}(c_I, \text{enc}(\langle x_3, n_I \rangle, k)) \\ P_R &:= \text{in}(c_R, y_1). \text{out}(c_R, \text{enc}(\langle y_1, n_R \rangle, k)). \\ &\quad \text{in}(c_R, y_2). \\ &\quad \text{let } y_3 = \text{eq}(y_2, \text{enc}(\langle n_R, y_1 \rangle, k)) \text{ in } 0 \end{aligned}$$

2.3 Semantics

The operational semantics of processes is given by a labeled transition system over *configurations* (denoted by K) which are pairs $(\mathcal{P}; \phi)$ where:

- \mathcal{P} is a multiset of ground processes where null processes are implicitly removed;
- $\phi = \{w_1 \mapsto u_1, \dots, w_n \mapsto u_n\}$ is a *frame*, *i.e.* a substitution where w_1, \dots, w_n are variables in \mathcal{W} , and u_1, \dots, u_n are messages.

We often write $P \cup \mathcal{P}$ instead of $\{P\} \cup \mathcal{P}$. The terms in ϕ represent the messages that are known by the attacker. Given a configuration K , $\phi(K)$ denotes its second component. Sometimes, we consider processes as configurations, in such cases, the corresponding frame is \emptyset .

IN	$(\text{in}(c, x).P \cup \mathcal{P}; \phi) \xrightarrow{\text{in}(c, R)} (P\{x \mapsto u\} \cup \mathcal{P}; \phi)$ where R is a recipe such that $R\phi \Downarrow u$ for some message u
OUT	$(\text{out}(c, u).P \cup \mathcal{P}; \phi) \xrightarrow{\text{out}(c, w)} (P \cup \mathcal{P}; \phi \cup \{w \mapsto u\})$ with w a fresh variable in \mathcal{W} .
LET	$(\text{let } \bar{x} := \bar{v} \text{ in } P \text{ else } Q \cup \mathcal{P}; \phi) \xrightarrow{\tau_{\text{then}}} (P\{\bar{x} \mapsto \bar{u}\} \cup \mathcal{P}; \phi)$ when $\bar{v} \Downarrow \bar{u}$ for some \bar{u}
LET-FAIL	$(\text{let } \bar{x} := \bar{v} \text{ in } P \text{ else } Q \cup \mathcal{P}; \phi) \xrightarrow{\tau_{\text{else}}} (Q \cup \mathcal{P}; \phi)$ when $v_i \nDownarrow$ for some $v_i \in \bar{v}$
NEW	$(\nu n.P \cup \mathcal{P}; \phi) \xrightarrow{\tau} (P \cup \mathcal{P}; \phi)$ where n is a fresh name from \mathcal{N}
REPL	$(!P \cup \mathcal{P}; \phi) \xrightarrow{\tau} (P \cup !P \cup \mathcal{P}; \phi)$
PAR	$(\{P_1 \mid P_2\} \cup \mathcal{P}; \phi) \xrightarrow{\tau} (\{P_1, P_2\} \cup \mathcal{P}; \phi)$

■ **Figure 2** Semantics for processes

The operational semantics of a process is given by the relation $\xrightarrow{\alpha}$ defined in Figure 2. The rules are quite standard and correspond to the intuitive meaning of the syntax given in the previous section. The first rule allows the attacker to send on channel c a message as soon as it is the result of a computation done by applying public function symbols on messages that are in his current knowledge. The second rule corresponds to the output of a term: the corresponding term is added to the frame of the current configuration, which means that the attacker gains access to it. The third and fourth rules correspond to the evaluation of a sequence of terms $\bar{v} = v_1, \dots, v_n$; if this succeeds, *i.e.* if there exist messages u_1, \dots, u_n such that $v_1 \Downarrow u_1, \dots, v_n \Downarrow u_n$ then variables \bar{x} are bound to those messages, and P is executed; otherwise the process will continue with Q . The three remaining rules allow one to execute a restriction, unfold a replication, and split a parallel composition.

The two first rules are the only observable actions. However, for reasons that will become clear later on, we make a distinction when a process evolves using LET or LET-FAIL.

► **Example 6.** Continuing Example 5. We have that: $P_{\text{Fh}} \xrightarrow{\text{tr}} (\emptyset; \phi_0)$ where tr and ϕ_0 are as follows, for fresh k' , n'_I and n'_R :

$$\text{tr} = \left\{ \begin{array}{l} \tau.\tau.\tau.\tau.\text{out}(c_I, w_1).\text{in}(c_R, w_1).\text{out}(c_R, w_2) \\ \text{in}(c_I, w_2).\tau_{\text{then}}.\text{out}(c_I, w_3).\text{in}(c_R, w_3).\tau_{\text{then}} \end{array} \right.$$

$$\phi_0 = \{w_1 \mapsto n'_I, w_2 \mapsto \text{enc}(\langle n'_I, n'_R \rangle, k'), w_3 \mapsto \text{enc}(\langle n'_R, n'_I \rangle, k')\}.$$

This execution corresponds to a normal execution of one session of the protocol.

The relation $\xrightarrow{\alpha_1 \dots \alpha_n}$ between configurations (where $\alpha_1 \dots \alpha_n$ is a sequence of actions) is defined as the transitive closure of $\xrightarrow{\alpha}$.

2.4 Trace equivalence

We are concerned with trace equivalence, which is commonly used [14, 21] to express many privacy-type properties such as anonymity, unlinkability, strong secrecy, etc. Intuitively, two configurations are trace equivalent if an attacker cannot tell whether he is interacting with one or the other. Before defining formally this notion, we first introduce a notion of equivalence between frames, called *static equivalence*.

- **Definition 7.** A frame ϕ is *statically included* in ϕ' when $\text{dom}(\phi) = \text{dom}(\phi')$, and
- for any recipe R such that $R\phi \Downarrow u$ for some u , we have that $R\phi' \Downarrow u'$ for some u' ;
 - for any recipes R_1, R_2 such that $R_1\phi \Downarrow u_1$, $R_2\phi \Downarrow u_2$, and $u_1 =_{\text{E}} u_2$, we have that $R_1\phi' \Downarrow =_{\text{E}} R_2\phi' \Downarrow$, i.e. there exist v_1, v_2 such that $R_1\phi' \Downarrow v_1$, $R_2\phi' \Downarrow v_2$, and $v_1 =_{\text{E}} v_2$.
- Two frames ϕ and ϕ' are in *static equivalence*, written $\phi \sim \phi'$, if the two static inclusions hold.

Intuitively, an attacker can distinguish two frames if he is able to perform some computation (or a test) that succeeds in ϕ and fails in ϕ' (or the converse).

- **Example 8.** Consider the frame ϕ_0 as given in Example 6, we have that $\phi_0 \sqcup \{w_4 \mapsto k'\} \not\sim \phi_0 \sqcup \{w_4 \mapsto k''\}$. Indeed, the attacker may observe that the computation $R = \text{dec}(w_2, w_4)$ succeeds in $\phi_0 \sqcup \{w_4 \mapsto k'\}$ but fails in $\phi_0 \sqcup \{w_4 \mapsto k''\}$.

Then, *trace equivalence* is the active counterpart of static equivalence taking into account the fact that the attacker may interfere during the execution of the process in order to distinguish between the two situations. Given a configuration $K = (\mathcal{P}; \phi)$, we define $\text{trace}(K)$:

$$\text{trace}(K) = \{(\text{tr}, \phi') \mid (\mathcal{P}, \phi) \xrightarrow{\text{tr}} (\mathcal{P}'; \phi') \text{ for some configuration } (\mathcal{P}'; \phi')\}.$$

We define $\text{obs}(\text{tr})$ to be the subsequence of tr obtained by erasing all the τ actions (i.e. $\tau, \tau_{\text{then}}, \tau_{\text{else}}$).

- **Definition 9.** Let K and K' be two configurations. We say that K is *trace included* in K' , written $K \sqsubseteq K'$, when, for any $(\text{tr}, \phi) \in \text{trace}(K)$ there exists $(\text{tr}', \phi') \in \text{trace}(K')$ such that $\text{obs}(\text{tr}) = \text{obs}(\text{tr}')$ and $\phi \sim \phi'$. They are in *trace equivalence*, written $K \approx K'$, when $K \sqsubseteq K'$ and $K' \sqsubseteq K$.

- **Example 10.** We may be interested in checking whether $K = (!P_{\text{Fh}}; \emptyset)$ and $K' = (!\nu k. (!\nu n_I. P_I \mid !\nu n_R. P_R); \emptyset)$ are in trace equivalence. Intuitively, this equivalence models the fact that P_{Fh} is unlinkable: each session of the protocol appears to an attacker as if it has been initiated by a different tag, since a given tag can perform at most one session in the idealized scenario K . This equivalence actually holds. It is non-trivial, and cannot be established using existing verification tools such as ProVerif or Tamarin. The technique developed in this paper will notably allow one to establish it automatically.

3 Our class of protocols and properties

We aim to propose sufficient conditions to ensure unlinkability and anonymity for a generic class of 2-party protocols. In this section, we define formally the class of protocols and the security properties we are interested in.

3.1 A generic class of 2- party protocols

As already mentioned, we consider 2-party protocols that are therefore made of two roles called the initiator and responder role respectively.

► **Definition 11.** An *initiator role* is a ground process obtained using the following grammar:

$$P_I ::= 0 \mid \text{out}(c, u).P_R$$

where $c \in \mathcal{C}$, $u \in \mathcal{T}(\Sigma_c, \mathcal{N} \cup \mathcal{X})$, and P_R is obtained from the grammar of *responder roles*:

$$\begin{aligned} P_R &::= 0 \\ &\mid \text{in}(c, y).\text{let } \bar{x} = \bar{v} \text{ in } P_I \text{ else } 0 \\ &\mid \text{in}(c, y).\text{let } \bar{x} = \bar{v} \text{ in } P_I \text{ else } \text{out}(c', u') \end{aligned}$$

where $c, c' \in \mathcal{C}$, $y \in \mathcal{X}$, \bar{x} (resp. \bar{v}) is a (possibly empty) sequence of variables in \mathcal{X} (resp. terms in $\mathcal{T}(\Sigma, \mathcal{N} \cup \mathcal{X})$), and $u' \in \mathcal{T}(\Sigma_c, \mathcal{N} \cup \mathcal{X})$.

Intuitively, a role describes the actions performed by an agent. A responder role consists of waiting for an input and, depending on the outcome of a number of tests, the process will continue by sending a message, or stop possibly outputting an error message. An initiator behaves similarly but begins with an output. The grammar forces to add a conditional after each input. This is not a real restriction as it is always possible to add trivial conditionals with empty \bar{x}, \bar{v} .

► **Example 12.** Continuing our running example, P_I (resp. P_R) as defined in Example 5 is an initiator (resp. responder) role (up to the addition of trivial conditionals).

Then, a protocol consists of an initiator role and a responder role that go together. This is formally stated through the notion of *honest trace*.

► **Definition 13.** A trace tr (*i.e.* a sequence of actions) is *honest* for a frame ϕ if $\tau_{\text{else}} \notin \text{tr}$ and $\text{obs}(\text{tr})$ is of the form: $\text{out}(_, w_0).\text{in}(_, R_0).\text{out}(_, w_1).\text{in}(_, R_1).\dots$ for arbitrary channel names, and such that $R_i\phi \Downarrow =_{\mathbf{E}} w_i\phi \Downarrow$ for any action $\text{in}(_, R_i)$ occurring in tr .

An honest trace is a trace in which the attacker does not really interfere, and that allows the execution to progress without going into an *else* branch that intuitively correspond to a way to abort the protocol.

Now, among the names that occur in the roles, we need to distinguish those that correspond to long-term data *e.g.* keys (called *identity names*) from others that are freshly generated at each session (called *session names*). We also need to introduce the notion of *public messages*. A message u is *public* if $u =_{\mathbf{E}} v$ for some $v \in \mathcal{T}(\Sigma_c \cap \Sigma_{\text{pub}}, \emptyset)$. Intuitively, a message is public if it is equal modulo \mathbf{E} to a term that is built using public symbols only.

► **Definition 14.** A protocol Π is a tuple $(\bar{k}, \bar{n}_I, \bar{n}_R, \mathcal{I}, \mathcal{R})$ where $\bar{k}, \bar{n}_I, \bar{n}_R$ are three disjoint sets of names, \mathcal{I} (resp. \mathcal{R}) is an initiator (resp. responder) role such that $\text{fn}(\mathcal{I}) \subseteq \bar{k} \sqcup \bar{n}_I$ (resp. $\text{fn}(\mathcal{R}) \subseteq \bar{k} \sqcup \bar{n}_R$). Names \bar{k} (resp. $\bar{n}_I \sqcup \bar{n}_R$) are called *identity names* (resp. *session names*).

Let $P_{\Pi} = \nu \bar{k}.(\nu \bar{n}_I.\mathcal{I} \mid \nu \bar{n}_R.\mathcal{R})$. We assume in addition that $P_{\Pi} \xrightarrow{\text{tr}_h} (\emptyset; \phi_h)$ for some frame ϕ_h that does not contain any public message, and some trace tr_h that is honest for ϕ_h .

► **Example 15.** Let $\Pi = (k, n_I, n_R, P_I, P_R)$ with P_I and P_R as defined in Example 5. We have already seen that P_I is an initiator role whereas P_R is a responder role. Let $P_{\Pi} = \nu k.(\nu n_I.P_I \mid \nu n_R.P_R)$. Let $\text{tr}_h = \text{tr}$, and $\phi_h = \phi_0$ as defined in Example 6. They satisfy the requirements stated in Definition 14, and therefore Π is a protocol according to our definition.

3.2 Security properties under study

We consider both anonymity and unlinkability as defined in [4]. Before recalling the formal definition of these two notions, we first introduce some useful notation.

Given a protocol Π , as defined above, we denote \mathcal{M}_Π the process that represents an arbitrary number of agents that may possibly execute an arbitrary number of sessions, whereas \mathcal{S}_Π represents an arbitrary number of agents that can at most execute one session each. Formally, we define:

$$\begin{aligned}\mathcal{M}_\Pi &:= !\nu\bar{k}.(!\nu\bar{n}_I.\mathcal{I} \mid !\nu\bar{n}_R.\mathcal{R}); \text{ and} \\ \mathcal{S}_\Pi &:= !\nu\bar{k}.(\nu\bar{n}_I.\mathcal{I} \mid \nu\bar{n}_R.\mathcal{R}).\end{aligned}$$

This allows us to formally state the notion of unlinkability that is studied in this paper.

► **Definition 16** (unlinkability). Let $\Pi = (\bar{k}, \bar{n}_I, \bar{n}_R, \mathcal{I}, \mathcal{R})$ be a protocol. We say that Π preserves *unlinkability* if $\mathcal{M}_\Pi \approx \mathcal{S}_\Pi$.

Informally, a protocol preserves unlinkability w.r.t. the roles \mathcal{I} and \mathcal{R} if each session of these roles looks to an outside observer as if it has been executed with different identity names. In other words, an ideal version of the protocol with respect to unlinkability, would allow the roles \mathcal{I} and \mathcal{R} to be executed at most once for each identity names. An outside observer should then not be able to tell the difference between the original protocol and the ideal version of this protocol. Although unlinkability of only one role (*e.g.* the tag for RFID protocols) is often considered in the literature, we consider a stronger notion where both roles are treated symmetrically.

In order to express anonymity w.r.t. some identities $\bar{id} \subseteq \bar{k}$, we introduce the following process: $\mathcal{M}_\Pi^{\text{id}} := \mathcal{M}_\Pi \mid \nu\bar{k}.(!\nu\bar{n}_I.\mathcal{I}_0 \mid !\nu\bar{n}_R.\mathcal{R}_0)$ where $\mathcal{I}_0 = \mathcal{I}\{\bar{id} \mapsto \bar{id}_0\}$, $\mathcal{R}_0 = \mathcal{R}\{\bar{id} \mapsto \bar{id}_0\}$, and \bar{id}_0 are fresh constants from $\Sigma_c \cap \Sigma_{\text{pub}}$ (*i.e.* not used in Π). In this process, in addition to the arbitrary number of agents that may execute an arbitrary number of sessions, there are two agents \mathcal{I}_0 and \mathcal{R}_0 that have disclosed (part of) their identity \bar{id}_0 to the attacker, and that may also execute an arbitrary number of sessions.

► **Definition 17** (anonymity). Let $\Pi = (\bar{k}, \bar{n}_I, \bar{n}_R, \mathcal{I}, \mathcal{R})$ be a protocol, and $\bar{id} \subseteq \bar{k}$. We say that Π preserves *anonymity w.r.t. \bar{id}* if $\mathcal{M}_\Pi \approx \mathcal{M}_\Pi^{\text{id}}$.

Defined in this way, anonymity ensures that an attacker does not see the difference between the system $\mathcal{M}_\Pi^{\text{id}}$ (in which \bar{id}_0 is present) and the original system \mathcal{M}_Π (in which \bar{id}_0 is not present). Since \bar{id}_0 is not present in the system \mathcal{M}_Π , his anonymity is trivially preserved.

4 Our Approach

We now define our two conditions, namely frame opacity and well-authentication, and our main result which states that these conditions are sufficient to ensure unlinkability and anonymity. Before doing that, we shall introduce annotations in the semantics of our processes, in order to ease their analysis. After having stated our conditions and result, we will illustrate that our conditions are realistic on various case studies.

4.1 Annotations

We shall now define an annotated semantics whose transitions are equipped with more informative actions. The annotated actions will feature labels identifying which concurrent

process has performed the action. This will allow us to identify which specific agent (with some specific identity and session names) performed some action.

Formally, an *annotation* is of the form $A(\bar{k}, \bar{n})$ where $A \in \{I, R\}$. An *annotated action* is either τ or $\alpha[a]$ where α is an action other than τ (possibly τ_{then} or τ_{else}) and a is an annotation. Finally, an *annotated process* is of the form $P[a]$ where P is a role process and a is an annotation.

Given a protocol $\Pi = (\bar{k}, \bar{n}_I, \bar{n}_R, \mathcal{I}, \mathcal{R})$, consider any execution of $\mathcal{M}_{\Pi}^{\text{id}}$, \mathcal{M}_{Π} or \mathcal{S}_{Π} . In such an execution, τ actions are solely used to instantiate new agents, by unfolding a replication, breaking a parallel and choosing fresh names. Performing these actions results in the creation of agents, that is, instances of \mathcal{I} and \mathcal{R} with fresh names. Actions other than τ (that is, input, output and conditionals) are then only performed by those agents.

This allows us to define an *annotated semantics* for our processes of interest. In that semantics, agents in the multiset of processes are annotated by their identity, and actions other than τ are annotated with the identity of the agent responsible for that action. Traces of the annotated semantics will be denoted by **ta**.

► **Example 18.** Considering the protocol of Example 15, process \mathcal{S}_{Π} can essentially perform the execution seen in Example 6. The annotated execution has the trace **ta** given below, where k' , n'_I and n'_R are fresh names, $a_I = I(k', n'_I)$ and $a_R = R(k', n'_R)$:

$$\begin{aligned} \mathbf{ta} = & \tau.\tau.\tau.\tau.\tau.\text{out}(c_I, w_1)[a_I].\text{in}(c_R, w_1)[a_R].\text{out}(c_R, w_2)[a_R].\text{in}(c_I, w_2)[a_I].\tau_{\text{then}}[a_I]. \\ & \text{out}(c_I, w_3)[a_I].\text{in}(c_R, w_3)[a_R].\tau_{\text{then}}[a_R] \end{aligned}$$

After the initial τ actions, the annotated configuration is $(\{\mathcal{I}\sigma_I[a_I], \mathcal{R}\sigma_R[a_R], \mathcal{S}_{\Pi}\}; \phi_0)$ where $\sigma_I = \{k \mapsto k', n_I \mapsto n'_I\}$, and $\sigma_R = \{k \mapsto k', n_R \mapsto n'_R\}$. The structure is preserved for the rest of the execution of **ta**, with three processes in the multiset (until they become null), two of which remaining annotated with a_I and a_R .

Note that annotations of the specific agents whose identity contains constants $\overline{\text{id}_0}$ will contain those constants (*i.e.* they are of the form $A(\bar{k}, \bar{n})$ with $\overline{\text{id}_0} \subseteq \bar{k}$).

4.2 Frame opacity

In light of attacks based on leakage from messages where non-trivial relations between outputted messages are exploited by the attacker to trace an agent, our first condition will basically require that, in any execution, outputs are completely opaque to the attacker, indistinguishable from pure randomness. Formally, we define this notion by comparing a frame with an ideal version of it, which is essentially obtained by replacing each message of a frame by a fresh name. However, in order to obtain a reasonable condition, we must make an exception there for constructors which can be inverted (*e.g.* pairs, lists, XML data) which we call *transparent* and are often used in protocols without any inherent risk.

► **Definition 19.** A set of constructors $\Sigma_t \subseteq \Sigma_c \cap \Sigma_{\text{pub}}$ is said to be *transparent* if it satisfies the following conditions:

- for all $f \in \Sigma_t$ of arity n , and for all $1 \leq i \leq n$, there exists a recipe $R_i \in \mathcal{T}(\Sigma_{\text{pub}}, \{w\})$ such that for any message $u = f(u_1, \dots, u_n) \in \mathcal{T}(\Sigma_c, \mathcal{N})$, one has $R_i\{w \mapsto u\} \Downarrow v_i$ for some v_i such that $v_i =_{\text{E}} u_i$;
- symbols of Σ_t do not occur in the equations of **E**.

In the rest of our theoretical development, we assume an arbitrary transparent set Σ_t . Our results are stronger with a larger set, but still hold if some constructor symbols fail to be identified as transparent.

► **Example 20.** In the signature of Example 1, the largest set of transparent constructors is $\{\langle \rangle, 0, \text{ok}\}$.

We now define the idealization of (the observable parts of) messages and frames: we first replace non-transparent sub-terms by holes (denoted by \square), and then fill-in these holes using distinct fresh names.

► **Proposition 1.** There exists a function $[\cdot]^{\text{ideal}} : \mathcal{T}(\Sigma_c, \mathcal{N}) \rightarrow \mathcal{T}(\Sigma_t, \{\square\})$ such that $[u]^{\text{ideal}} = f([u_1]^{\text{ideal}}, \dots, [u_n]^{\text{ideal}})$ whenever $u =_E f(u_1, \dots, u_n)$ for $f \in \Sigma_t$, and $[u]^{\text{ideal}} = \square$ otherwise. Furthermore, we have that $[u]^{\text{ideal}} = [v]^{\text{ideal}}$ whenever $u =_E v$.

► **Definition 21.** A *concretization* of $u_t \in \mathcal{T}(\Sigma_t, \{\square\})$ is any term obtained by replacing each hole of u_t by a fresh nonce. We denote by $\text{inst}(u_t)$ the set of all concretizations of u_t . Finally, for a message u , we let $[u]^{\text{nonce}}$ be the set $\text{inst}([u]^{\text{ideal}})$.

Those definitions are extended to frames in the natural way, with the freshness condition on nonces being understood at the level of frame and not of individual messages. As a result, we immediately have that, for any $u' \in [u]^{\text{nonce}}$ (resp. $\phi' \in [\phi]^{\text{nonce}}$), no nonce appears twice in u' (resp. ϕ'), and therefore for all frames ψ and $\phi_1, \phi_2 \in [\psi]^{\text{nonce}}$, one has $\phi_1 \sim \phi_2$.

► **Example 22.** Let u be $\langle n_P, \text{enc}(\langle \text{ok}, n_P \rangle, k) \rangle$. We have that $[u]^{\text{ideal}} = \langle \square, \square \rangle$ and $[u]^{\text{nonce}} = \{\langle n_1, n_2 \rangle \mid n_1 \neq n_2 \in \mathcal{N}\}$.

We are now ready to state our first condition:

► **Definition 23.** A process P ensures *frame opacity* when for any $(P; \emptyset) \xrightarrow{\text{tr}} (Q; \phi)$, we have $\phi \sim \psi$ for some $\psi \in [\phi]^{\text{nonce}}$. Given a protocol Π and some names $\bar{i}d \subseteq \bar{k}$, we say that Π ensures *frame opacity* when $\mathcal{M}_{\Pi}^{\text{id}}$ does.

► **Example 24.** Consider the frame ϕ_0 as defined in Example 6. We have that

$$[\phi_0]^{\text{ideal}} = \{w_1 \mapsto \square, w_2 \mapsto \square, w_3 \mapsto \square\}.$$

We have that $\phi_0 \sim \phi$ for any $\phi \in [\phi_0]^{\text{nonce}}$. However, in case, $n'_R = n'_I$, static equivalence between ϕ_0 and its idealized version $[\phi_0]^{\text{nonce}}$ does not hold, and therefore any protocol that generates such a frame is not frame opaque.

4.3 Well-authentication

Our second condition will prevent the attacker to obtain some information about agents through the outcome of conditionals. To do so, we will essentially require that conditionals of \mathcal{I} and \mathcal{R} can only be executed successfully in honest, intended interactions. It is unnecessary to impose such a condition on conditionals that never leak any information, which are found in several security protocols. We characterize below a simple class of such conditionals, for which the attacker will always know the outcome of the conditional based on the past interaction.

► **Definition 25.** A conditional $\text{let } \bar{x} = \bar{v} \text{ in } P \text{ else } Q$ occurring in $\mathcal{A} \in \{\mathcal{I}, \mathcal{R}\}$ is *safe* if $\bar{v} \in \mathcal{T}(\Sigma_{\text{pub}}, \{x_1, \dots, x_n\} \cup \{u_1, \dots, u_n\})$, where the x_i are the variables bound by the previous inputs of that role, and u_i are the messages used in the previous outputs of that role.

► **Example 26.** Consider the process given below:

$$\text{out}(c, u). \text{in}(c, x). \text{let } z = \text{neq}(x, u) \text{ in } P \text{ else } Q$$

The conditional is used to ensure that the agent will not accept as input the message he sent at the previous step. Such a conditional is safe according to our definition.

Note that trivial conditionals the grammar forced us to add are safe and will thus not get in the way of our analysis.

We can now formalize the notion of association, which expresses that two agents are having an honest, intended interaction (*i.e.* the attacker essentially did not interfere in their communications). For an annotated trace \mathbf{ta} and annotations a and a' , we denote by $\mathbf{ta}|_{a,a'}$ the subsequence of \mathbf{ta} that consists of actions of the form $\alpha[a]$ or $\alpha[a']$.

► **Definition 27.** Two agents $A_1(\bar{k}_1, \bar{n}_1)$ and $A_2(\bar{k}_2, \bar{n}_2)$ are *associated* in (\mathbf{ta}, ϕ) if:

- the agents are *dual*, *i.e.* $A_1 \neq A_2$ and $\bar{k}_1 = \bar{k}_2$;
- the interaction $\mathbf{ta}|_{A_1(\bar{k}_1, \bar{n}_1), A_2(\bar{k}_2, \bar{n}_2)}$ is honest for ϕ .

► **Example 28.** Continuing Example 18, the agents $I(k', n'_I)$ and $R(k', n'_R)$ are associated in (\mathbf{ta}, ϕ_0) .

We can finally state our second condition:

► **Definition 29.** The protocol Π is *well-authenticating* if, for any execution

$$(\mathcal{M}_{\Pi}^{\text{id}}; \emptyset) \xrightarrow{\mathbf{ta}, \tau_{\text{then}}[A(\bar{k}, \bar{n}_1)]} (\mathcal{P}; \phi)$$

either the last action corresponds to a safe conditional, or there exists A' and \bar{n}_2 such that $A(\bar{k}, \bar{n}_1)$ and $A'(\bar{k}, \bar{n}_2)$ are associated in (\mathbf{ta}, ϕ) , and $A'(\bar{k}, \bar{n}_2)$ is only associated with $A(\bar{k}, \bar{n}_1)$ in (\mathbf{ta}, ϕ) .

Intuitively, this condition does not require anything for safe conditional as we already know that they cannot leak new information to the attacker (he already knows their outcome). For unsafe conditionals, it requires that whenever an agent a evaluates them positively (*i.e.* he does not abort the protocol), it must be the case that this agent a is so far having an honest interaction with a dual agent a' . Indeed, as discussed in Introduction, it is crucial to avoid such unsafe conditionals to be evaluated positively when the attacker is interfering because this could leak crucial information.

The last requirement of the definition is related to the fact that an agent may be associated with several agents (which would systematically break unlinkability); this can in fact only happen when this agent has not performed any input yet. This requirement is in fact equivalent to imposing that the first input of the responder \mathcal{R} is not immediately followed by an unsafe test. Indeed, at this stage of the interaction, the initiator has not performed an input yet, and may thus be associated to other responder agents. Conversely, the requirement becomes obvious once the responder has sent a message to the initiator.

4.4 Soundness w.r.t. unlinkability and anonymity

Our main theorem establishes that the previous two conditions are sufficient to ensure unlinkability and anonymity:

► **Theorem 30.** Consider a protocol $\Pi = (\bar{k}, \bar{n}_I, \bar{n}_R, \mathcal{I}, \mathcal{R})$ and some identity names $\bar{id} \subseteq \bar{k}$. If the protocol is well-authenticating and ensures frame opacity, then Π ensures unlinkability and anonymity w.r.t. \bar{id} .

Note that, since $\mathcal{M}_{\Pi}^{\text{id}} \approx \mathcal{M}_{\Pi}$ when $\bar{id} = \emptyset$, we have as a corollary that if \mathcal{M}_{Π} ensures well-authentication and frame opacity, then Π is unlinkable.

4.5 Practical Impact

We first briefly discuss how to delegate the verification of frame-opacity and well-authentication to a fully automatic tool. It is actually possible to use ProVerif[27] to do so. Well-authentication is basically a conjunction of reachability properties, which can be checked in ProVerif using correspondence properties [3]. Assuming well-authentication, it is possible to check frame opacity using the diff-equivalence feature of ProVerif [12]. We implemented all translations needed to leverage ProVerif in our tool UKano [29]. This tool basically takes as inputs a specification of a protocol in our class and, by applying some translations and by calling ProVerif, it automatically checks our two conditions (and thus unlinkability and anonymity).

Let us comment on its practical impact. We summarize the result of the confrontation of our method to our case studies in Figure 3, focusing on unlinkability. We remark that our conditions have proven to be tight enough for all our case studies: when a condition fails to hold, we could always discover a real attack on unlinkability. Most of the positive results (when unlinkability holds) and all attacks are new. Note that all positive results were established automatically using our tool UKano (based on ProVerif) and all ProVerif files can be found at [29].

Protocol	Frame opacity	Well auth.	Unlink.
Feldhofer	✓	✓	safe
Hash-Lock [23]	✓	✓	safe
LAK [23] (stateless)	–	×	attack
Fixed LAK	✓	✓	safe
BAC [4]	✓	✓	safe
BAC [4]/PA/AA	✓	✓	safe
PACE [26] (faillible dec)	–	×	attack
PACE (as <i>e.g.</i> in [10])	–	×	attack
PACE [26]	–	×	attack

■ **Figure 3** Summary of our case studies. We note ✓ for a condition automatically checked using our tool UKano (based on ProVerif) and × when the condition does not hold. PACE (faillible dec.) refers to the protocol as described in the official specification [26] when using a symmetric decryption that may fail (when used with wrong keys). Finally, PACE as in the official specification [26] with a non-faillible dec is also linkable w.r.t. our definition. However, this attack does not translate to a practical attack in the very specific context of ePassport but may be exploited in other contexts.

5 Conclusion

We have identified two conditions, namely well-authentication and frame opacity, which imply anonymity and unlinkability for a wide class of protocols. Additionally, these conditions can be checked automatically using the tool ProVerif. This yields a new verification technique to check anonymity and unlinkability for an unbounded number of sessions. It has proved quite effective on various case studies. In particular, it has brought first-time unlinkability proofs for the BAC e-passport protocol, but also combinations of these protocols and the companion PA and AA protocols. Our case studies also illustrated that our methodology is useful to discover attacks against unlinkability and anonymity.

In the future, we plan to develop a mature implementation of our tool implementing this

technique, in order to make it widely accessible for the design and study of privacy-preserving two-party protocols. We also plan to seek for more case studies and apply our verification method to them. We could also try to translate our conditions into more comprehensive guidelines helping the design of new privacy-enhancing protocols.

We also identify a number of research problems aimed at generalizing the impact of our technique. Currently, our conditions are checked using *ProVerif* which, despite its great flexibility, supports only a limited kind of equational theory. In particular, associative-commutative theories needed for xor (widely used in RFID protocols) are not supported. It seems likely that frame opacity can be checked using ad-hoc methods rather than *ProVerif*, which could support wider classes of theories. Concerning well-authentication, we could consider various extensions of *ProVerif* with partial support for xor [24], or other tools such as *Tamarin* and *Maude-NPA*. We also would like to investigate the extension of our main theorem to the case of protocols with state. This is certainly technically challenging, but would make it possible to model more protocols, or at least model them more faithfully.

References

- 1 Iso 15408-2: Common criteria for information technology security evaluation - part 2: Security functional components, July 2009.
- 2 M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proceedings of POPL'01*. ACM Press, 2001.
- 3 Martín Abadi and Bruno Blanchet. Computer-assisted verification of a protocol for certified email. In *Static Analysis*, pages 316–335. Springer, 2003.
- 4 Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *Proceedings of CSF'10*. IEEE Comp. Soc. Press, 2010.
- 5 A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for Google apps. In *Proc. 6th ACM Workshop on Formal Methods in Security Engineering (FMSE'08)*, pages 1–10. ACM, 2008.
- 6 A. Armando et al. The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In *Proc. 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*, volume 7214, pages 267–282. Springer, 2012.
- 7 Michael Backes, Catalin Hritcu, and Matteo Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, 23-25 June 2008*, pages 195–209. IEEE Computer Society, 2008.
- 8 David Basin, Jannik Dreier, and Ralf Sasse. Automated symbolic proofs of observational equivalence. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1144–1155. ACM, 2015.
- 9 Mathieu Baudet. Deciding security of protocols against off-line guessing attacks. In *Proc. 12th Conference on Computer and Communications Security*. ACM, 2005.
- 10 Jens Bender, Marc Fischlin, and Dennis Kügler. Security analysis of the pace key-agreement protocol. In *Information Security*, pages 33–48. Springer, 2009.
- 11 B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of CSFW'01*, pages 82–96. IEEE Comp. Soc. Press, 2001.
- 12 Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. In *Logic in Computer Science, 2005. LICS 2005. Proceedings. 20th Annual IEEE Symposium on*, pages 331–340. IEEE, 2005.

- 13 Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 2008.
- 14 Mayla Bruso, K. Chatzikokolakis, and J. den Hartog. Formal verification of privacy for RFID systems. In *Proceedings of CSF'10*, 2010.
- 15 Vincent Cheval and Bruno Blanchet. Proving more observational equivalences with proverif. In *Principles of Security and Trust*, pages 226–246. Springer, 2013.
- 16 Vincent Cheval, Hubert Comon-Lundh, and Stéphanie Delaune. Trace equivalence decision: Negative tests and non-determinism. In *Proceedings of CCS'11*. ACM Press, 2011.
- 17 Rémy Chrétien, Véronique Cortier, and Stéphanie Delaune. From security protocols to pushdown automata. *ACM Transactions on Computational Logic*, 17(1:3), September 2015.
- 18 V. Cortier and B. Smyth. Attacking and fixing Helios: An analysis of ballot secrecy. *Journal of Computer Security*, 21(1):89–148, 2013.
- 19 Véronique Cortier, Stéphanie Delaune, and Pascal Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 14(1):1–43, 2006.
- 20 Stéphanie Delaune, Steve Kremer, and Mark D. Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, (4), 2008.
- 21 Stéphanie Delaune, Steve Kremer, and Mark D. Ryan. Verifying privacy-type properties of electronic voting protocols: A taster. In *Towards Trustworthy Elections – New Directions in Electronic Voting*, volume 6000. Springer, 2010.
- 22 Martin Feldhofer, Sandra Dominikus, and Johannes Wolkerstorfer. Strong authentication for rfid systems using the aes algorithm. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 357–370. Springer, 2004.
- 23 Ari Juels and Stephen A Weis. Defining strong privacy for rfid. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):7, 2009.
- 24 Ralf Küsters and Tomasz Truderung. Reducing protocol analysis with XOR to the xor-free case in the horn theory based approach. *J. Autom. Reasoning*, 46(3-4):325–352, 2011.
- 25 S. Meier, B. Schmidt, C. Cremers, and D. Basin. The Tamarin Prover for the Symbolic Analysis of Security Protocols. In *Proc. 25th International Conference on Computer Aided Verification (CAV'13)*, volume 8044 of *LNCS*, pages 696–701. Springer, 2013.
- 26 Technical advisory group on machine readable travel documents (tag/mrtd).
- 27 Proverif: Cryptographic protocol verifier in the formal model.
- 28 Sonia Santiago, Santiago Escobar, Catherine Meadows, and José Meseguer. A formal definition of protocol indistinguishability and its verification using maude-npa. In *Security and Trust Management*, pages 162–177. Springer, 2014.
- 29 Webpage hosting our tool ukano, our case studies and the full paper: <http://projects.lsv.ens-cachan.fr/ukano>.
- 30 Ton Van Deursen and Sasa Radomirovic. Attacks on rfid protocols. *IACR Cryptology ePrint Archive*, 2008:310, 2008.

Learning and Verifying Unwanted Behaviours

Wei Chen¹, David Aspinall¹, Andrew D. Gordon^{1,2}, Charles Sutton¹, and Igor Muttik³

¹ University of Edinburgh, UK,
`{wchen2,csutton}@inf.ed.ac.uk`,
`{David.Aspinall,Andy.Gordon}@ed.ac.uk`

² Microsoft Research Cambridge, UK,

³ Intel Security, UK,
`igor.muttik@intel.com`

Abstract. Unwanted behaviours, such as interception and forwarding of incoming messages, have been repeatedly seen in Android malware. We study the problem of learning unwanted behaviours from malware instances and verifying the application in question to deny these behaviours. We approximate an application’s behaviours by an automaton, i.e., finite control-sequences of events, actions, and annotated API calls, and develop an efficient machine-learning-centred method to construct and choose abstract sub-automata, to characterise unwanted behaviours exhibited in hundreds and thousands of malware instances. By taking the verification results against unwanted behaviours as input features, we show that the performance of detecting new malware is improved dramatically, in particular, the precision and recall are respectively 8% and 51% better than those using API calls and permissions, which are the best performing features known so far. This is the first automatic approach to generate unwanted behaviours for machine-learning-based Android malware detection. We also demonstrate unwanted behaviours constructed for well-known malware families. They compare well to those described in human-authorised descriptions of these families.

Keywords: mobile security, static analysis, software verification, machine learning, malware detection

1 Introduction

Android malware, including trojans, spyware and other kinds of unwanted software, has been increasingly seen in the wild and even on official app stores [2, 4, 22, 45]. Researchers and malware analysts have organised malware instances into hundreds of families [37, 45], e.g., Basebridge, Geinimi, Ginmaster, Spitmo, Zitmo, etc. These malware instances share certain unwanted behaviours, for example, sending premium messages constantly, collecting personal information, loading classes from hidden payloads then executing commands from remote servers, and so on. Except some inaccurate online analysis reports [1, 3, 5, 6, 32] of identified malware families, however, people have no idea of what exactly happens in these malware instances.

We want to learn unwanted behaviours from hundreds and thousands of malware instances and verify the application in question to deny them. We will show that these unwanted behaviours can improve the classification performance of detecting new malware. Our approach is outlined as follows.

- **Formalisation.** We approximate an Android application’s behaviours by a finite-state automaton, that is, finite control-sequences of events, actions, and annotated API calls. Since different API calls might indicate the same behaviour, we abstract the automaton by aggregating API calls into permission-like phrases. We call it a *behaviour automaton*.
- **Learning.** An *unwanted behaviour* is a common behaviour which is shared by malware instances and has been rarely seen in benign applications. We develop a machine-learning-centred method to infer unwanted behaviours, by efficiently constructing and selecting sub-automata from behaviour automata of malware instances. This process is guided by the behavioural difference between malware and benign applications.
- **Refinement.** To purify unwanted behaviours, we exploit the family names of malware instances to help figure out the most informative unwanted behaviours. We compare unwanted behaviours with the human-authorised descriptions for malware families, to ensure that they match well with patterns described in these descriptions.
- **Verification.** We check whether the application in question has any security fault by verifying whether the intersection between its behaviour automaton and an unwanted behaviour is not empty.

To simulate new malware detection, we take malware instances released in different years and collected from different sources respectively as training, validation and testing sets. We use a group of malware and benign applications released before 2014 as the training and validation sets. Malware instances in these sets were collected from Malware Genome Project [45] and Mobile-Sandbox [37]. We take a collection of malware and benign applications released in 2014 as the testing set. They were supplied by Intel Security. We use API calls, permissions, and the verification results against unwanted behaviours as input features; then apply L1-Regularized Linear Regression [30, 39] to train classifiers. The evaluation on the testing set shows that the precision and recall of using unwanted behaviours are respectively 8% and 51% better than those of using API calls and permissions, which are the best performing input features known so far for machine-learning-based Android malware detectors. As shown in Table 2, using API calls and permissions as input features, can achieve very good precision and recall on the validation set, however, its classification performance on the testing set is poor. That is, unwanted behaviours are more general than API calls and permissions. This is needed in practice, to mitigate over-fitting and improve the robustness of malware classifiers.

Our approach is the first to automatically generate unwanted behaviours for Android malware classification. The main contributions of this paper are:

- We show that it is hard to detect new malware for classifiers trained on identified malware. We demonstrate that by using semantics-based features

like unwanted behaviours, the classification performance of detecting new malware is dramatically improved.

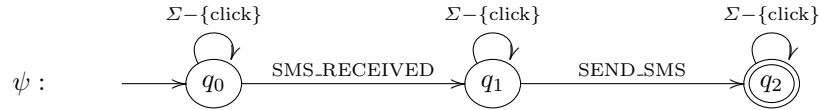
- We have designed and implemented a static analysis tool to construct behaviour automata from the byte-code of Android applications, with regard to a broad range of features of the Android framework.
- The complexity of constructing all sub-automata is exponential in the number of malware instances. We propose and apply a new machine-learning-centred algorithm to combat this.
- We develop a refinement approach to look up the most informative unwanted behaviours, by making use of the family names of malware instances.

2 An Example Unwanted Behaviour

Let us consider a malware family called Ggtracker. A brief description of this family, which was produced by Symantec [6], is as follows.

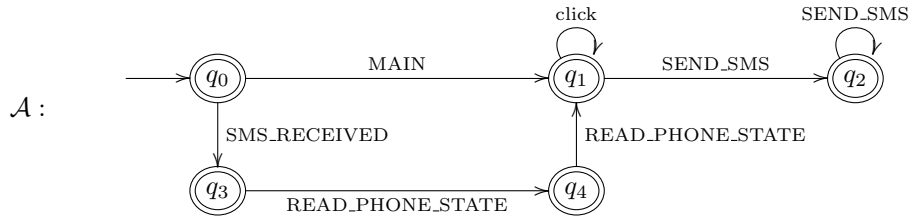
It sends SMS messages to a premium-rate number. It monitors received SMS messages and intercepts SMS messages. It may also steal information from the device.

One of unwanted behaviours we have learned from malware instances in this family can be expressed as the regular expression: `SMS_RECEIVED.SEND_SMS`. The approach to learn these unwanted behaviours will be elaborated in Sections 4 and 5. It denotes the behaviour of sending an SMS message out *immediately* after an incoming SMS message is received. We generalise from this unwanted behaviour and construct the following automaton.



Here, we use the symbol Σ to denote the collection of events, actions, and permission-like phrases and the word “click” to denote that there is no interaction from the user. This automaton formalises the unwanted behaviour of sending an SMS message out after an incoming SMS message is received with no interaction from the user. It is actually a set of super-sequences of the sequence `SMS_RECEIVED.SEND_SMS`.

We now want to verify whether a target application has the above unwanted behaviour. Let us consider the following behaviour automaton \mathcal{A} . It is constructed from the byte-code of an Android application. Its source code and the method to construct behaviour automata will be given in Section 3.



It tells us: this application has two entries which are respectively specified by actions MAIN and SMS_RECEIVED; it will collect information like your phone state, then send SMS messages out; the behaviour of sending SMS messages can also be triggered by a user interaction, e.g., click a button, touch the screen, long-press a picture, etc., which is denoted by the word “click”. All states in this automaton are accepting states since any prefix of an application’s behaviours is one of its behaviours as well.

Because the intersection between \mathcal{A} and ψ is not empty, we consider this application is unsafe with respect to the unwanted behaviour ψ . In Section 6, we will show that this verification against unwanted behaviours can improve the classification performance of new malware detecting.

3 Behaviour Automata

We use a simplified synthetic application to illustrate the construction of behaviour automata. This application will constantly send out the device ID and the phone number by SMS messages on background when an incoming SMS message is received. Its source code and part of its manifest file are as follows.

```
/* Main.java */
public class Main extends
    Activity implements View.OnClickListener {
    private static String info = "";
    protected void onCreate(Bundle savedInstanceState) {
        Intent intent = getIntent();
        info = intent.getStringExtra("DEVICE_ID");
        info += intent.getStringExtra("TEL_NUM");
        Task task = new Task();
        task.execute(); }
    public void onClick (View v) {
        Task task = new Task();
        task.execute(); }
    private class Task extends AsyncTask<Void, Void, Void> {
        protected Void doInBackground(Void... params) {
            while (true) {
                SmsManager sms = SmsManager.getDefault();
                sms.sendTextMessage("1234", null, info, null, null); }
            return null; }}}
```

```
/* Receiver.java */
public class Receiver extends BroadcastReceiver {
    public void onReceive(Context context, Intent intent) {
        Intent intent = new Intent();
        intent.setAction("com.main.intent");
        TelephonyManager tm = (TelephonyManager)
            getBaseContext().getSystemService(Context.TELEPHONY_SERVICE);
        intent.putExtra("DEVICE_ID", tm.getDeviceId());
        intent.putExtra("TEL_NUM", tm.getLine1Number());
        sendBroadcast(intent); }}
```

```
/* AndroidManifest.xml */
<activity android:name="com.example.Main" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <action android:name="com.main.intent" />
    </intent-filter>
</activity>
<receiver android:name="com.example.Receiver" >
    <intent-filter>
```

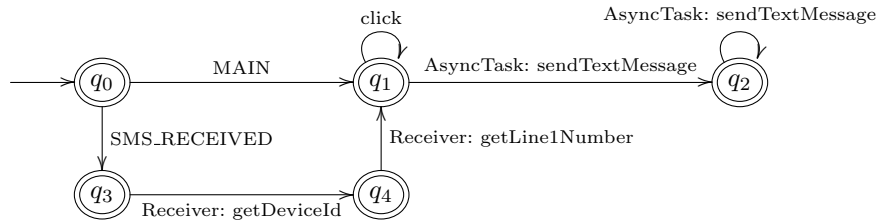
```

    <action android:name="android.provider.Telephony.SMS_RECEIVED" />
  </intent-filter>
</receiver>

```

As specified in `AndroidManifest.xml`, the Main activity can handle a specific Intent called “com.main.intent” and the Receiver will be triggered by an incoming SMS message (SMS_RECEIVED). After the Receiver collects the device ID and the phone number, it will send them out by a broadcast with the intent “com.main.intent”. This broadcast is then handled by the Main activity in the method `onCreate`. Afterwards, SMS messages containing the device ID and the phone number are sent out on background in an `AsyncTask`.

We compile this application. From its byte-code we construct the following automaton.



This automaton is a collection of finite control-sequences of actions, events, and annotated API calls. Actions reflect what happens in the environment and what kind of service an application requests for, e.g., an incoming message is received, the device finishes booting, the application wants to send an email by using the service supplied by an email-client, etc. Events denote the interaction from the user, e.g., clicking a picture, pressing a button, scrolling down the screen, etc. Annotated API calls tell us whether the application is doing anything we are interested in. For instance, `getDeviceID`, `getLine1Number`, and `sendTextMessage` are annotated API calls in the above example.

Notice that for a single behaviour there are often several related API methods. For example, `getDeviceId`, `getLine1Number`, and `getSimSerialNumber` are all related to the behaviour of reading phone state. We want to categorise API methods into a set of phrases, which describe behaviours of applications, so as to remove redundancy caused by API calls which indicate the same behaviour. This results in an abstract automaton, so-called a *behaviour automaton*. It has several advantages, including: more resilient to variants of behaviours, such as swapping two API calls related to the same behaviour; more compact than the original automata, which is good for human-understanding and further analysis, by reducing the number of labels on the edges. For instance, the behaviour automaton for the above example is the automaton \mathcal{A} depicted in Section 2.

To construct such a behaviour automaton directly from an Android application, we have modelled complex real-world features of the Android framework, including: inter-procedural calls, callbacks, component life-cycles, permissions, actions, events, inter-component communications, multiple threads, multiple entries, nested classes, interfaces, and runtime-registered listeners. We don’t model registers, fields, assignments, operators, comparison, pointer-aliases, arrays or

exceptions. The choice of which features to model is a trade-off between efficiency and precision. Automata are much more accurate than the manifest information, e.g., permissions and actions, which were often used as input features for malware detection or mitigation [12, 23, 25]. Compared with a simple list of API calls appearing in code, an automaton can capture more sophisticated behaviours. This is needed in practice, because: API calls appearing in code contain “noise” caused by dead code and libraries [7]; and, some unwanted behaviours only arise when some API methods are called in certain orders [17, 31, 43]. On the other hand, automata are less accurate than models which capture data-flows. However, it is much easier to generate behaviour automata using our tool for applications en masse than generating data-flows using tools like FlowDroid [10] or Amandroid [41]. In particular, people can annotate appealing API methods to generate compact behaviour automata more efficiently, rather than considering all data-dependence between statements.

In our implementation, we use an extension of permission-governed API methods generated by PScout [11] as annotations. The Android platform tools `aapt` and `dexdump` are respectively used to extract the manifest information and to decompile byte-code into assembly code, from which we construct the automaton. It took around two weeks to generate automata for 10,000 applications using a multi-core desktop computer.

4 Learning Unwanted Behaviours

Once a behaviour automaton has been constructed for each malware instance, we want to capture the common behaviour shared by malware, which is rarely seen in benign applications, so-called an *unwanted behaviour*. The space of candidate behaviours, which consists of the intersection and difference between behaviour automata, in theory, is exponential in the number of sample applications. To combat this, we approximate this space by searching for a “salient” subspace. The searching process is guided by the behavioural difference between malware and benign applications. We formalise this process as the following algorithm.

Function: `construct_features` (G, α)

Input: G – a group of behaviour automata

α – the lower bound on the classification accuracy

Output: salient sub-automata and their weights

```

1:  $G_{i \in [0..N-1]} \leftarrow$  divide the set  $G$  into  $N$  groups
2: for  $i \in [0..N-1]$ 
3:    $F_i \leftarrow$  merge_features ( $G_i, \emptyset$ )
4:  $s \leftarrow 2$ 
5: while  $s \leq N$ 
6:   for  $i \in [0..N-1]$ 
7:      $j \leftarrow i - (s/2)$ 
8:     if  $(i+1) \% s = 0$  then
9:        $(F_i, -), (F_j, -) \leftarrow$  diff_features ( $F_i, \alpha$ ), diff_features ( $F_j, \alpha$ )
10:       $F_i \leftarrow$  merge_features ( $F_i, F_j$ )
11:    elif  $(i+1) > (N/s) \times s$  and  $(i+1) \% (s/2) = 0$  then
```

```

12:    $(F_i, -), (F_j, -) \leftarrow \text{diff\_features}(F_i, \alpha), \text{diff\_features}(F_j, \alpha)$ 
13:    $s \leftarrow s \times 2$ 
14:   return  $\text{diff\_features}(F_{s/2-1}, \alpha)$ 
Function: merge_features( $E, F$ )
1:   for  $e \in E$ 
2:     for  $f \in F$ 
3:       if  $f - e \neq \emptyset$  then  $F \leftarrow F \cup \{f - e\}$ 
4:       if  $f \cap e \neq \emptyset$  then  $F \leftarrow F \cup \{f \cap e\}$ 
5:       if  $f - e \neq f$  and  $f \cap e \neq f$  then  $F \leftarrow F - \{f\}$ 
6:        $e \leftarrow e - f$ 
7:       if  $e \neq \emptyset$  then  $F \leftarrow F \cup \{e\}$ 
8:   return  $F$ 
Function: diff_features( $F, \alpha$ )
1:    $D \leftarrow$  add an equal number of randomly-chosen benign applications
2:   into the set of malware instances from which  $F$  was collected
3:    $W, \text{acc} \leftarrow \text{train}(D, F)$ 
4:   if  $\text{acc} > \alpha$  then  $F \leftarrow \{f \in F \mid W_f \neq 0\}$ 
5:   return  $F, W$ 

```

The main process `construct_features` takes a collection G of behaviour automata as input and outputs a set F of salient sub-automata with their weights W . Here, a sub-automaton is *salient* if it is actually used in a linear classifier, i.e., its weight is not zero.

We randomly divide G into N groups: $G_0, \dots, G_i, \dots, G_{N-1}$. For each group, we construct sub-automata by computing the intersection and difference between automata within this group, i.e., `merge_features` (G_i, \emptyset). This results in N feature sets $F_0, \dots, F_i, \dots, F_{N-1}$. The sub-automata in each set are disjoint. Then, we merge sub-automata from different groups, i.e., `merge_features` (G_i, G_j). This process stops until all groups have been merged into a single group.

Before merging sub-automata from two different groups, for each group, we train a linear classifier, i.e., `train` (D, F), using a training set D and a feature set F . This training set consists of behaviour automata of malware instances in the group and an equal number of behaviour automata of randomly-chosen benign applications. The input feature set F consists of disjoint sub-automata, which are constructed from behaviour automata of malware instances in the group. Then, if the classification accuracy acc on the training set is above a lower bound α , we return sub-automata with non-zero weights. Otherwise, we return all features in F . This process differentiates salient features by adding benign applications. It is formalised as the function `diff_features`.

In our implementation, we adopt L1-Regularized Logistic Regression [30, 39] as the training method. This is because this method is specially designed to use fewer features. We set the lower bound α on the classification accuracy to 90%. We have also designed and implemented a multi-process program to accelerate the construction, i.e., construct sub-automata for each group simultaneously. It took around one week to process 4,000 malware instances using a multi-core desktop computer. At the end of the computation, we produced around 1,000 salient sub-automata.

We will use these salient sub-automata to characterise unwanted behaviours. A straightforward way is to choose automata by their weights, for example, those with negative weights, i.e., $\{f \in F \mid W_f < 0\}$. More complex methods to capture unwanted behaviours will be discussed in next section.

5 Refining Unwanted Behaviours

To purify unwanted behaviours, we want to exploit the family names of malware instances to figure out the most informative ones, that is, to choose a small set of salient sub-automata to characterise unwanted behaviours for each family. Here are several candidate methods:

- *Top- n -negative*. For a linear classifier, intuitively, a feature with a negative weight more likely indicates an unwanted behaviour, and a feature with a positive weight more likely indicates a normal behaviour. This observation leads us to refine unwanted behaviours by using sub-automata with negative weights, i.e., choose the top- n features from the set $\{f \in F \mid W_f < 0\}$ by ranking the absolute values of their weights.
- *Subset-search*. For each malware family, we choose a subset X of salient sub-automata, such that it largely covers and is strongly associated with malware instances in this family. Formally, we use $Pr(f|X)$ to denote the probability of a malware instance belonging to a family f if all automata in X are sub-automata of the behaviour automaton of this instance, and $Pr(X|f)$ to denote the probability of all automata in X are sub-automata of the behaviour automaton of a malware instance if this instance belongs to f . We use F_1 -measure as the evaluation function to look up subsets. i.e., $\frac{2Pr(f|X)Pr(X|f)}{Pr(f|X)+Pr(X|f)}$. Since exhaustively searching a power-set space is expensive, we adopt Beam Search [33, Chapter 6] to approximate the best K -subsets.
- *TF-IDF*. Another method is to consider features as terms, features from malware instances in a family as a document, and the multi-set of features as the corpus. We rank features by their TF-IDF (term frequency and inverse document frequency) and choose a maximum of m features to characterise unwanted behaviours of each family.

We collected more than 4,000 malware instances from Malware Genome Project [1, 45] and Mobile-Sandbox [9, 37]. They have been manually investigated and organised into around 200 families by third-party researchers and malware analysts. We collected human-authorised descriptions for these families from their online analysis reports [1, 3, 5, 6, 32].

We then produce salient sub-automata from these malware instances by applying the algorithm in previous section and construct unwanted behaviours for each family by combining all methods discussed earlier. We list manual descriptions and learned unwanted behaviours of 10 prevalent families in Table 1.

A subjective comparison shows that these learned unwanted behaviours compare well to their manual descriptions. Also, they reveal trigger conditions of

<i>manual description</i>	<i>learned unwanted behaviours in regular expressions</i>
<i>Arspam. Sends spam SMS messages to contacts on the compromised device [6].</i>	1. <code>BOOT_COMPLETED.SEND_SMS</code>
<i>Anserverbot. Downloads, installs, and executes payloads [1].</i>	1. <code>UMS_CONNECTED.LOAD_CLASS*.(ACCESS_NETWORK_STATE READ_PHONE_STATE INTERNET).(ACCESS_NETWORK_STATE READ_PHONE_STATE INTERNET LOAD_CLASS)*</code>
<i>Basebridge. Forwards confidential details (SMS, IMSI, IMEI) to a remote server [3]. Downloads and installs payloads [1, 6].</i>	1. <code>UMS_CONNECTED.(INTERNET LOAD_CLASS READ_PHONE_STATE ACCESS_NETWORK_STATE)+</code>
<i>Cosha. Monitors and sends certain information to a remote location [6].</i>	1. <code>MAIN.click.(click ACCESS_FINE_LOCATION DIAL)*.DIAL.(click ACCESS_FINE_LOCATION DIAL)*.(INTERNET ϵ)</code> 2. <code>SMS_RECEIVED.(INTERNET ACCESS_FINE_LOCATION)+</code>
<i>Droiddream. Gains root access, gathers information (device ID, IMEI, IMSI) from an infected mobile phone and connects to several URLs in order to upload this data [1, 3].</i>	1. <code>PHONE_STATE.(ACCESS_NETWORK_STATE READ_PHONE_STATE+ INTERNET).(ACCESS_NETWORK_STATE INTERNET)*</code>
<i>Geinimi. Monitors and sends certain information to a remote location [6]. Introduces botnet capabilities with clear indications that command and control (C&C) functionality could be a part of the Geinimi code base [5].</i>	1. <code>ϵ MAIN.click+.VIBRATE.(click VIBRATE)*.RESTART_PACKAGES.(MAIN.(click VIBRATE)*.RESTART_PACKAGES)*</code> 2. <code>BOOT_COMPLETED.(ACCESS_NETWORK_STATE click INTERNET RESTART_PACKAGES ACCESS_FINE_LOCATION)+</code>
<i>Ggtracker. Monitors received SMS messages and intercepts SMS messages [3]</i>	1. <code>MAIN.READ_PHONE_STATE</code> 2. <code>SMS_RECEIVED.SEND_SMS</code>
<i>Ginmaster. Sends received SMS messages to a remote server [32]. Downloads and installs applications without user concern [32].</i>	1. <code>BOOT_COMPLETED.LOAD_CLASS</code> 2. <code>MAIN.SEND_SMS</code>
<i>Spitmo. Filters SMS messages to steal banking confirmation codes [6].</i>	1. <code>NEW_OUTGOING_CALL.READ_PHONE_STATE.INTERNET.(INTERNET ϵ)</code>
<i>Zitmo. Opens a backdoor that allows a remote attacker to steal information from SMS messages received on the compromised device [6].</i>	1. <code>SMS_RECEIVED.SEND_SMS</code> 2. <code>MAIN.READ_PHONE_STATE</code> 3. <code>MAIN.SEND_SMS</code>

Table 1. Learned unwanted behaviours versus manual descriptions.

some behaviours, which were often lacking in manual descriptions. For example, the expression `BOOT_COMPLETED.SEND_SMS` denotes that after the device finishes booting, this application will send a message out; the expression `UMS_CONNECTED.LOAD_CLASS` means that when a USB mass storage is connected to the device, this application will load some code from a library or a hidden payload; and the unwanted behaviour for Droiddream shows that if the phone state changes (`PHONE_STATE`), this application will collect information then access Internet. Within manual descriptions displayed in Table 1, only two behaviours are not captured by learned unwanted behaviours: “gain root access” for Droiddream and the behaviour of Spitmo.

6 Evaluation: Detecting New Malware

We are concerned with whether unwanted behaviours can help improve the robustness of malware classification. As we will show in Table 2, a linear classifier using API calls and permissions as input features, which are popular input features for Android malware detectors [7, 9, 12, 15, 29, 44], performs badly on new malware instances (the testing set), although it has a very good classification performance on the validation set. In this section, we will show that unwanted behaviours improve the classification performance of new malware detection.

We collected 3,000 malware instances, which have been discovered before 2014, and 3,000 randomly-chosen benign applications. They include some famous benign applications, such as Google Talk, Amazon Kindle, and Youtube, and so on; and all malware instances from Malware Genome Project [1, 45] and most malware instances from Mobile-Sandbox [9, 37]. These malware instances have been manually investigated and organised into around 200 families by third-party researchers and malware analysts. By reading their online malware analysis reports [1, 3, 5, 6, 32], we learned what bad things would happen in these malware instances. We divided them into a training set and a validation set. Each of them consists of 1,500 malware instances across all families and 1,500 benign applications.

We test using a collection of 1,500 malware instances, which were released in 2014, and 1,500 randomly-chosen benign applications. These malware instances were from Intel Security and have been investigated by malware analysts. But, there is no family information or online analysis report about them. We have no idea of unwanted behaviours of these malware instances.

Permissions and lists of API calls appearing in code are extracted from these applications as input features to train classifiers as baselines.

We construct behaviour automata for all applications, then apply methods discussed in Sections 4 and 5 to learn unwanted behaviours from malware instances in the training set. Some behaviours of the application in question are not the same as unwanted behaviours, but, they often have unwanted behaviours as sub-sequences. For example, although the word `SMS_RECEIVED.SEND_SMS` is not accepted by the automaton \mathcal{A} in Section 2, \mathcal{A} accepts some sequences containing this word as a subsequence, i.e., `SMS_RECEIVED.READ_PHONE_`

STATE.READ_PHONE.STATE.SEND_SMS⁺. To capture behaviours sharing the same patterns with unwanted behaviours, if a behaviour contains an unwanted behaviour as a sub-sequence, we consider this behaviour as unwanted as well. We call them *extended* unwanted behaviours. We check whether the intersection between the behaviour automaton of the application in question and an (extended) unwanted behaviour is not empty. We collect these verification results as input features to train the target classifiers.

For both baselines and target classifiers, we use L1-Regularized Logistic Regression [30, 39] as the training method.

feature training (2011–13)	validation (2011–13)		testing (2014)		#salient/#feature
	precision	recall	precision	recall	
signature-based features (baselines)					
permissions	89%	99%	53%	21%	59/175
apis	91%	98%	61%	15%	1443/52432
apis & permissions	93%	98%	65%	15%	735/52607
semantics-based features (targets)					
unwanted behaviours	66%	91%	53%	74%	634/886
ext. unwanted	75%	87%	69%	66%	581/886
ext. unwanted for families	72%	72%	73%	66%	131/131
mixed features					
all	95%	99.5%	65%	7.5%	870/61149

Table 2. Classification performance using different features.

The classification performance is reported in Table 2. It confirms that:

- Unwanted behaviours dramatically improve the classification performance on new malware instances. The classification performance using API calls and permissions as input features is very good on the validation set, i.e., the precision and recall are respectively 93% and 98%. However, this is just over-fitting to the training set, since its performance on the testing set is bad, in particular, the precision is 65% and recall is 15%. This means that a lot of new unwanted behaviours cannot be captured by API calls and permissions. By using the verification results against unwanted behaviours as input features, we improve the precision to 73% and the recall to 66%, as shown in the row of “ext. unwanted for families”.
- Refining unwanted behaviours using the family names helps improve the classification performance of detecting new malware. The precision is increased from 69% (in the row of “ext. unwanted”) to 73% (in the row of “ext. unwanted for families”), while maintaining the same recall. This refinement also helps reduce the number of features which are actually used in a linear classifier, in particular, totally 131 features were used, rather than 581 features.

7 Related Work

Our approach is close to the methodology proposed by Fredrikson et al. to synthesize malware specification [27]. In their work, a data dependence graph with logic constraints on nodes and edges was used to characterise an app’s behaviours. From graphs of malware instances and benign apps they constructed so-called significant subgraphs that maximise the information gain. Then, several optimal collections of subgraphs were selected as specifications by using the formal concept analysis. They produced 19 specifications using 166 subgraphs constructed from 534 malware instances in 6 families and 39 benign apps. The evaluation was done on a collection consisting of 378 new malware instances and 28 benign apps. The main drawback of this method is its scalability. Also, the training and testing sets are very unbalanced, i.e., the number of benign apps is much less than that of malware instances.

Our approach is more scalable by using behaviour automata rather than data-flow models to approximate the behaviours of apps. Instead of using graph mining and formal concept analysis, we explore the weights assigned by the linear classifiers to accelerate the searching for salient sub-automata and refine the learned unwanted behaviours by exploiting the family names. These techniques enable our approach to deal with hundreds and thousands of malware instances spread in hundreds of families. Also, we tested on balanced datasets with equal numbers of malware instances and benign apps. By doing this the precision and recall are more comparable and convincing.

Machine learning methods have been applied to automatically detect Android malware [7, 9, 12, 15, 20, 28, 29, 38, 44]. DroidAPIMiner [7] uses refined API calls as features and relies on the KNN (k -nearest neighbours) algorithm. The method Drebin [9] extracts a broad range of features, such as permissions, components, API calls, and intents, from the manifest file and disassembled code, then trains a SVM classifier. The classifier produced by Yerima et al. [44] extracts permissions, API calls, and commands as features. Another interesting tool is CHABADA [29] which detects outliers (abnormal API usage) within clusters of applications by exploiting OC-SVM (one-class SVM). These clusters were grouped by descriptions of applications using LDA (Latent Dirichlet Allocation). Among others, the tool Dendroid [38] uses the cosine similarity between vectors of call graphs of malware to help group unknown malware samples into identified families. Similar ideas were applied in DroidLegacy [20] to detect piggybacking by exploiting the difference between API sets of modules in malware instances belonging to different families. Another interesting tool MAST [15] exploits MCA (Multiple Correspondence Analysis) to figure out indicative features. All of these tools and methods were trying to obtain good fits to a dataset by using different methods and variant kinds of features. The robustness of malware classification, in particular, the classifier specifically designed for new malware detection, has received much less consideration.

Aside from machine learning methods, static and dynamic analysis were applied to help malware detection in Android applications. Enck et al. [23] explored combinations of permissions to mitigate malware. Felt et al. [25] detected

over-privilege of permissions. Tools like TaintDroid [21], CopperDroid [34], and MonitorMe [31] are able to monitor dynamic behaviours of applications. The tool ComDroid [18] was designed for the detection of intent-based vulnerabilities. The tool Apposcopy extracts the weighted API dependency graphs to improve the classification performance [26]. More sophisticated tools like FlowDroid [10] and Amandroid [41] can identify paths from sources to sinks of sensitive information. All of these tools were designed to identify (or eliminate) malware (or unwanted behaviours) without learning or summarising useful properties for verification from their analysis results.

The idea of characterising applications’ behaviours as automata is similar with the behaviour abstraction in [13, 42]. The behaviour automata are close to permission-event graphs [17], embedded call graphs [28], and behaviour graphs [43]. But, none of them has been exploited to automatically generate verifiable properties.

The unwanted behaviours can be considered as instances of security automata [35]. Our verification approach is the same as the automata-theoretic model checking [40]. More sophisticated approaches and finer formalisations can be found in the study of LTL model checking, e.g., Song et. al.’s work [36]. Totally 19 malicious properties for Android applications were manually constructed and specified as the first-order LTL formulae in [31]. Some benign and malicious properties specified in LTL were verified against hundreds of Android applications in [17]. But, none of these properties was automatically constructed.

Among others, Angluin’s [8] and Biermann’s [14] algorithms were developed to learn regular expressions from sample finite strings. To apply similar ideas in unwanted behaviour construction, we have to extract enough finite strings from applications. Comparing with the construction of behaviour automata, this will be more expensive.

8 Conclusion and Further Work

To learn compact, natural, and verifiable unwanted behaviours from Android malware instances is challenging and has not yet been considered. Compared with manually-composed properties, unwanted behaviours, which are automatically constructed from malware instances, will be much easier to be updated on the changes of behaviours exhibited in new malware instances. To the best of our knowledge, our approach is the first to automatically construct temporal properties from Android malware instances. We show that unwanted behaviours help improve the classification performance, in particular, they dramatically increase the precision and recall of detecting new malware. These unwanted behaviours can not only be used to eliminate potentially new instances of known malware families but also help people’s understanding of unwanted behaviours exhibited in these families.

Some unwanted behaviours cannot be captured by our formalisation, e.g., gain root access, and some are not captured precisely enough, e.g., botnet controls. In further work, we want to extend the current formalisation to capture

more sophisticated behaviours precisely. We will also try to combine the output of dynamic analysis, e.g., traces produced by CopperDroid [34] or MonitorMe [31], with that of static analysis to approximate applications' behaviours. We will explore whether properties expressed in LTL are needed in the practice of malware detection and whether it is possible to learn them from malware instances. The verification method adopted in this paper is straightforward and simple. More efficient and complex methods, e.g., the method discussed in [36] and model checking pushdown systems [24], will be considered in future. Except unwanted behaviours, we will investigate whether other methods can help improve the robustness of malware classifiers, e.g., semi-supervised learning [16]. On the other hand, since unwanted behaviours are context-sensitive, i.e., an unwanted behaviour in a group of applications might be normal in another, we want to organise applications into groups sharing similar behaviours and construct unwanted behaviours for each group. Also, in practice, many malware instances have no family information. We will explore methods to organise malware instances when the family information is unavailable, e.g., clustering by compression [19]. It is also interesting to study whether unwanted behaviours can convince people of the automatic malware detection.

References

1. Malware Genome Project. <http://www.malgenomeproject.org/>, 2012.
2. Mobile Adware and Malware Analysis. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/madware_and_malware_analysis.pdf, 2013.
3. Forensic Blog. <http://forensics.spreitzenbarth.de/android-malware/>, 2014.
4. McAfee Mobile Security Report. <http://www.mcafee.com/us/resources/reports/rp-mobile-security-consumer-trends.pdf>, 2014.
5. Juniper Networks. https://www.juniper.net/security/auto/includes/mobile_signature_descriptions.html, 2015.
6. Symantec security response. http://www.symantec.com/security_response/, 2015.
7. Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *SecureComm*, pages 86–103. Springer, 2013.
8. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, Nov. 1987.
9. D. Arp et al. Drebin: Efficient and explainable detection of Android malware in your pocket. *NDSS*, pages 23–26, 2014.
10. S. Arzt et al. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, pages 259–269, 2014.
11. K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android permission specification. In *CCS*, pages 217–228, 2012.
12. D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In *CCS*, pages 73–84, 2010.
13. P. Beaucamps, I. Gnaedig, and J.-Y. Marion. Behavior abstraction in malware analysis. In *Runtime Verification*, LNCS 6418, pages 168–182. Springer, 2010.

14. A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, June 1972.
15. S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: Triage for market-scale mobile malware analysis. In *WiSec*, pages 13–24, 2013.
16. O. Chapelle, B. Schölkopf, and A. Zien. *Semi-Supervised Learning*. The MIT Press, 1st edition, 2010.
17. K. Z. Chen et al. Contextual policy enforcement in Android applications with permission event graphs. In *NDSS*, 2013.
18. E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *MobiSys*, pages 239–252, 2011.
19. R. Cilibrasi and P. M. B. Vitnyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51:1523–1545, 2005.
20. L. Deshotels, V. Notani, and A. Lakhota. DroidLegacy: Automated familial classification of Android malware. In *PPREW*, 2014.
21. W. Enck et al. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM*, 57(3):99–106, 2014.
22. W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *USENIX Security Symposium*, 2011.
23. W. Enck, M. Ongtang, and P. D. McDaniel. On lightweight mobile phone application certification. In *CCS*, pages 235–245, 2009.
24. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification*, LNCS 1855, pages 232–247. Springer, 2000.
25. A. P. Felt et al. Android permissions demystified. In *CCS*, pages 627–638, 2011.
26. Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *SIGSOFT FSE*, 2014.
27. M. Fredrikson et al. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP ’10, pages 45–60, 2010.
28. H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of Android malware using embedded call graphs. In *AISec*, pages 45–54, 2013.
29. A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *ICSE*, 2014.
30. T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, 2001.
31. J.-C. Kuester and A. Bauer. Monitoring real android malware. In *Runtime Verification 2015*, Vienna, Austria, sep 2015.
32. McAfee Threat Center. <http://www.mcafee.com/uk/threat-center.aspx>, 2015.
33. P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers Inc., 1st edition, 1992.
34. A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct Android malware behaviors. In *European Workshop on System Security (EUROSEC)*, 2013.
35. F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, Feb. 2000.
36. F. Song and T. Touili. LTL model-checking for malware detection. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 416–431. 2013.
37. M. Spreitzerbarth, T. Schreck, F. Ehtler, D. Arp, and J. Hoffmann. Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security*, 14(2):141–153, 2015.

38. G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. *Expert Systems with Applications*, 41(4, Part 1):1104 – 1117, 2014.
39. R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.
40. M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *J. Comput. Syst. Sci.*, 32(2):183–221, 1986.
41. F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *CCS*, pages 1329–1341. ACM, 2014.
42. J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. *SIGSOFT Softw. Eng. Notes*, 27(4):218–228, July 2002.
43. C. Yang et al. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In *ESORICS*, September 2014.
44. S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik. A new Android malware detection approach using bayesian classification. In *AINA*, pages 121–128, 2013.
45. Y. Zhou and X. Jiang. Dissecting Android malware: characterization and evolution. In *IEEE Symposium on Security and Privacy*, pages 95–109, 2012.

Towards Safe Enclaves

Neline van Ginkel Raoul Strackx Jan Tobias Mühlberg
Frank Piessens

iMinds-DistriNet, KU Leuven, Celestijnenlaan 200A, B-3001 Belgium

Abstract

Protected module architectures, like the recently launched Intel Software Guard Extensions (Intel SGX), make it possible to protect individual software modules of an application against attacks from other modules of the application, or from the operating system. But if the code of the protected module (the *enclave* in Intel SGX terminology) has vulnerabilities itself, that module can still be exploitable. Programming enclaves in a safe programming language can prevent a wide range of vulnerabilities within the enclave.

However, the simple approach of programming enclaves in a safe programming language gives less security guarantees than one might expect. A safe language only provides safety guarantees for whole programs, not for individual modules that are part of a bigger program. If the context of the module is malicious, additional defensive measures are required to guarantee the safety of the enclave. This paper illustrates this problem, and reports on work-in-progress towards a solution.

1 Introduction

Software systems are often attacked by exploiting low-level details of their implementation. Attacks that exploit memory safety errors are an obvious example [EYP10]: by triggering a memory safety bug in a C or C++ program, the program writes to memory cells that it is not supposed to write to, and the effect of this memory corruption depends on low-level details of the compiler, operating system and hardware. By carefully choosing inputs sent to the program, the attacker can often make the program misbehave.

But also more recent attacks like memory scraping [Huq15] are an example: an attacker that can compromise the operating system can exfiltrate secret information (e.g. keys or credit card information) by scanning the virtual memory space of applications running on the operating system.

An important defense against memory corruption attacks are *safe programming languages* like Java, Scala or Rust [MK14]. For an attacker model where the attacker can interact with a complete program by providing input and by

reading output, such safe languages provide complete protection against memory corruption. Roughly speaking, a safe language ensures this by making sure that behavior of programs is always well-defined whatever input the attacker provides, thus making it impossible for the program to run into *undefined* behavior that might be implementation-dependent and potentially dangerous.

An important defense against memory scraping attacks are *protected module architectures* [MPP⁺08, SP12, NAD⁺13], now also supported in recent Intel processors through the new Intel Software Guard Extensions, Intel SGX [Int14, MAB⁺13]. For an attacker model where the attacker can control all infrastructural software, including the operating system and language runtime libraries, a protected module architecture can execute a software component in a *protected module* (called *enclave* in Intel SGX). Execution as a protected module provides strong assurance that only a module’s code can access that module’s memory, thus countering memory scraping attacks [CBB⁺01, SYP⁺09] and many other layer-below attacks.

These two defense mechanisms, safe languages and protected module architectures, complement each other well: protected module architectures limit what a malicious *context* can do to a software module, and language safety avoids memory safety vulnerabilities *within* a module. This paper reports on work in progress on the combination of these two mechanisms. We show how a module that is programmed in a safe language and executes in an enclave can still be subject to low-level attacks that exploit implementation details and can cause memory corruption within the module. This is due to the fact that a software module within an enclave may offer a richer API than just input and output of primitive values (like integers or strings). Methods or functions callable from the malicious context might also return (or accept as parameters) references to mutable objects, or function pointers.

The objective of our work is to extend the notion of safety to handle these additional cases: we investigate what defensive measures a compiler must take to ensure that a protected module that is written in a safe language never runs into undefined behavior. Note that this problem is related to – but different from – *fully abstract* compilation [Aba99], and safe language interoperability [MF07]. We discuss the similarities and differences in Section 5.

The remainder of this paper is structured as follows: first, in Section 2, we briefly recap the two mechanisms, language safety and protected module architectures, which we combine in this paper. Section 3 shows that the straightforward combination of these two does not give us safe enclaves: enclaves programmed in a safe language can still be unsafe in the sense that they can run into undefined, hence implementation-dependent, behavior. Section 4 informally sketches the additional checks a compiler should do to ensure safety, and outlines the status of this work in progress. Finally, in Sections 5 and 6, we discuss related work and conclude.

We emphasize that this paper reports on *work in progress*. We illustrate problems and solutions informally. A full formalization and implementation of the ideas in this paper is future work.

2 Background

2.1 Safe languages

Safe languages, like Java or C#, avoid implementation-dependent behavior and hence also any kind of attack that exploits such implementation-dependent behavior, including memory corruption attacks, by ensuring that the execution of programs is always well-defined. Examples of program constructs that could lead to undefined behavior include (1) accessing an array out of bounds (a spatial memory safety error), (2) accessing memory that has been deallocated (a temporal memory safety error), or (3) treating a data pointer as a function pointer (a type safety error).

Safe languages use a combination of compiler-enforced bounds checks (to counter spatial memory safety errors), automatic memory management (to counter temporal memory safety errors) and type checking (to counter type safety errors) to make sure that no program can ever run into such undefined behavior. Many safe languages compile to a virtual machine (like the Java Virtual Machine), but there is currently a growing interest in more C-like languages that compile to machine code and give the programmer more control over memory management, while still providing substantial safety guarantees. A prototypical example is the Rust programming language [MK14], strongly influenced by the Cyclone language [JMG⁺02].

Programming language safety is well understood [Pie02], but new challenges arise if such languages are used to build protected modules, as we will show further in the paper.

For this work-in-progress report, we will use a simple safe single-threaded dialect of C that supports global variables that are module-private (like static global variables in C), top-level function definitions, and type safe function pointers (like C# delegates). The only types are integers, the void type and function types. Section 3 contains examples of programs in this language. Formalizing the syntax and semantics of the language is straightforward.

2.2 Protected Module Architectures

Protected Module Architectures [MPP⁺08, MLQ⁺10, SP12] are a relatively recent countermeasure to protect software modules against attacks from the software context in which a module executes. This context includes both other modules of the program as well as higher privileged infrastructural software such as the operating system. They have been designed both for higher-end processors [MPP⁺08, SP12] – Intel’s most recent Skylake processors provide support under the name of Intel Software Guard Extensions (Intel SGX) [Int14, MAB⁺13] – as well as for small micro-processors [SPP10, NAD⁺13, KSSV14, EDFPT12]. Essentially, a Protected Module Architecture provides hardware enforced memory access control that can be used to ensure that the state of a protected module can only be accessed by the code of that protected module.

As a small representative example, consider the program in Figure 1. The

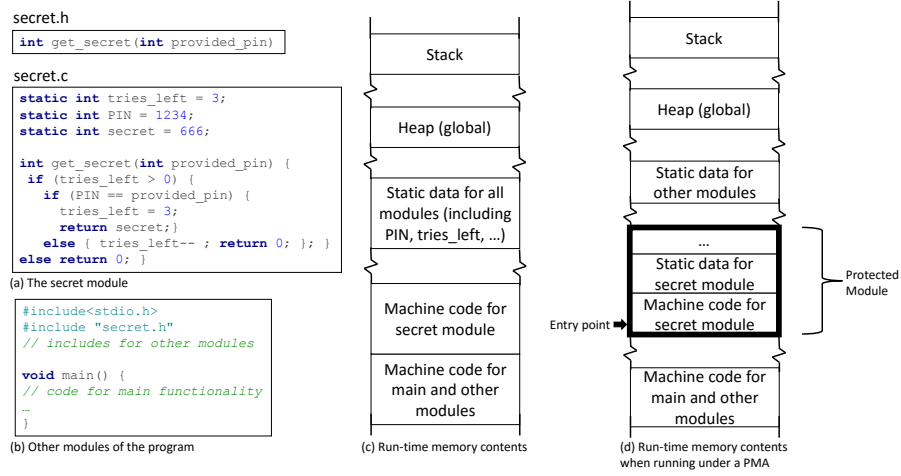


Figure 1: A program with a security critical module, and how it looks in memory at run-time with and without the protection of a Protected Module Architecture.

program has a single module (implemented in `secret.c`) that manages sensitive information, in this case the secret variable that should only be shown to users of the program who can provide a correct PIN. After three tries with an incorrect PIN, the module will refuse further attempts to protect against brute force attacks. The secret module exposes (in its header file `secret.h`) only the `get_secret()` function, and hence access to the global variables in `secret.c` is restricted to the secret module at source code level. Other modules (including for instance the `main()` function) can only interact with the module through the `get_secret()` function. This module is a very simple example of a security critical module, like for instance the password manager in a browser or the implementation of a cryptographic protocol. These modules can be subject to memory scraping attacks.

To see this, consider the compiled version of this program at run-time. Part (c) of Figure 1 shows a schematic picture of the memory contents at run-time. If we now consider an attacker that can compromise some of the other modules of the program, or that can compromise the operating system, it is clear that the attacker can easily violate both integrity as well as confidentiality of the variables of the secret module.

Part (d) of Figure 1 shows how the secret module could be loaded in a protected module (or enclave). A protected module is essentially a segment of memory, that can contain both code and data. In addition, the module has one or more *entry points*¹, that are code addresses in the protected module.

¹Some protected module architectures support only one entry point, but then multiple *logical* entry points can be layered on top of a single *physical* entry point.

The memory access control model of protected module architectures essentially enforces the following rules:

- When the Instruction Pointer (IP) is outside of the protected module, access to memory in the protected module is prohibited.
- When the IP is inside a protected module, access to memory within that module is allowed, but access to memory belonging to other protected modules is still prohibited.
- The only way for the IP to enter a protected module is by jumping to one of the designated entry points.

This simple access control model makes it possible for modules to guard access to their private state. As shown in Part (d) of Figure 1, the secret module could be compiled such that its machine code and its static data is stored within the protected module. Depending on the exact compilation algorithm, other state including for instance stack activation records for functions defined in the module can also be stored in the protected module. If we provide a single entry point to call the `get_secret()` function, then the variables `PIN`, `tries_left` and `secret` can only be accessed by the `get_secret()` function. Because of the memory access control, they can no longer be “scraped” from memory by malicious machine code in one of the other modules nor by kernel-level malware.

Given that Intel SGX is and will remain the dominant Protected Module Architecture for the foreseeable future, we will use the term “enclave” for the remainder of the paper.

3 Safe and unsafe modules

Safe languages protect against vulnerabilities *within* a module and Protected Module Architectures guard modules against attacks from *outside* of the module. Thus, it appears natural to combine both mechanisms. However, since the code in an enclave is typically only *part* of a program, and since safe languages only provide safety guarantees for whole programs written in that safe language, enclaves that are programmed in safe languages are *not* necessarily safe.

Consider the following example module in the simple safe language introduced in Section 2.1:

```
1 int count = 0;
2
3 void inc() {
4     count = count + 1;
5 }
6
7 int get_count() {
8     return count;
9 }
```

This implements a simple counter module where the context can only increment the counter, or read it out. If one compiles this module to an enclave in the way described in Section 2.2, the resulting module is safe in the sense that the context can never drive it into undefined behavior (assuming that the source code semantics specifies what should happen on integer overflow).

Now consider the following, somewhat more elaborate module:

```

1 int count = 0;
2 void (*obs)(int);
3
4 void set_observer(void o(int)) {
5     obs = o;
6 }
7
8 void inc() {
9     count = count + 1;
10    obs(count);
11 }
12
13 int get_count() {
14     return count;
15 }

```

If this module is compiled as outlined in Section 2.2, it is *not* safe. The entry point `set_observer()` accepts a function pointer as argument from the context. If the context is malicious, it can pass in an invalid function pointer, and then further behavior of the module will be undefined. By choosing the invalid function pointer carefully, an attacker could possibly succeed in making the module misbehave. Suppose for instance that the compiler represents function pointers as addresses of the start of the compiled code of the function. Then a malicious context can pass in any address within the module and have the module start executing code at that address on the call to `obs()`, thus launching a return-oriented-programming style attack [Sha07, CDD⁺10] against the module. For instance, if the module contains code like in Figure 1, such an attack could pass in a function pointer that points into the middle of the `get_secret()` function, and hence executes this function while skipping the validation tests.

Issues similar to the example above (where the context provides invalid function pointers), can also occur when the context passes in data pointers (the context could for instance tamper with the bounds information associated with a data pointer), or object references. Additional complications arise when code within the module reads from memory residing outside the module.

In this paper, we focus on the issue of handling invalid function pointers only to illustrate the essence of the problem.

4 Towards provably safe enclaves

Low-level attacks like the example above exist because of the abstraction gap between the machine code level and the source code level. The key to defending against these attacks is to make sure that any interaction that a malicious context could have with a compiled module (running in an enclave) at the machine code level will always have well-defined behavior according to the source code level semantics.

Proving such safety requires us to (1) model the context-enclave interactions at machine code level, (2) define the source code semantics modularly, and (3) show that *any* context-enclave interaction at machine code level can be interpreted as a source code level interaction for which the semantics defines corresponding behavior. We address these three items in the following three subsections.

4.1 Machine-code level context-enclave interactions

To ensure that behavior of an enclave is always well-defined, we have to make sure that *any* potential interaction the context could have with the enclave is covered by the definition of the semantics of the module that executes inside the enclave. For our simple model of enclaves, the following interactions are possible:

1. **jump-in** $e(v_i)$: jump from the context to an entry point e of the enclave, with values v_i in the processor registers. Assuming a 32-bit processor, both e and the v_i will be 32-bit words. i ranges from 0 to $N - 1$ where N is the number of registers in the processor, and v_i is the sequence of values in these registers at the time of the jump.
2. **jump-out** $a(v_i)$: jump from within the enclave to an address a outside of the module, with values v_i in the processor registers.
3. **read-out** $v = *a$: read from within the enclave the contents of memory at an address a outside of the module, resulting in value v .
4. **write-out** $*a = v$: write the value v from within the enclave to an address a outside of the module.

Only **jump-in** interactions are initiated by the context, the other three are initiated by the enclave. For simplicity, we will limit our attention in this paper to compilers that never perform **read-out** or **write-out** interactions. Handling them is left for future work; the key additional challenge is to make sure that data read from the context is checked as defensively as data passed into the module by means of a **jump-in** event. Under this simplifying assumption, the only possible interactions between enclave and context are **jump-in** or **jump-out** interactions.

An *enclave specification* is a labeled transition system, consisting of:

- A set ES of enclave states, the disjoint union of active enclave states AS (the processor is executing in the enclave) and passive enclave states PS (the processor is executing outside the enclave).
- A set of actions A , the disjoint union of entry actions **jump-in** $e(v_i)$, exit actions **jump-out** $a(v_i)$ and τ (the internal action).
- A transition relation $\subset ES \times A \times ES$, written $E \rightarrow^\alpha E'$, such that:
 - Entry actions transition from a passive state to an active state
 - Exit actions transition from an active state to a passive state
 - The internal action transitions from active states to active states

An enclave specification is *safe* (or *complete*), if the following holds:

- For any passive state P of the enclave, and for any entry action e , P transitions under e to some (necessarily active) state A .
- If P transitions under e to A , A never gets stuck, i.e. either infinitely performs τ actions, or will eventually perform an exit action.

Our objective is to make sure that a source code module defines a safe or complete enclave specification: the behavior of the enclave on any (even maliciously chosen) interaction with the context should follow from the source code semantics or lead to a well-defined error state.

Note that this notion of safety guarantees the absence of low-level vulnerabilities that are caused by undefined behavior, but it does *not* guarantee the absence of (for instance) undesired information leaks from the enclave, or it does not give any availability guarantees. We discuss this in more detail in Section 5.

4.2 A modular semantics of the source code

Since enclaves only contain a single software module, not necessarily a whole program, we have to define a *modular* semantics of the source language. A good example of such a modular semantics is the trace-based semantics of Java modules defined by Jeffrey and Rathke [JR05].

For our simple source language, we define the set of *values*² to be the disjoint union of (1) the signed 32-bit integers, (2) the value void, and (3) the set of function values. The set of function values consists of the disjoint union of (1) the function names of the functions defined in the module (the *internal* function values), and (2) an infinite set of abstract *external* function values.

The modular semantics defines the source code level interactions that the module can have with its source code context, and defines the traces of such interactions that are considered valid behaviors of the module. For instance, interactions for our simple language will be: (1) calls from the context of functions

²Note that we are now defining values at the level of the source code. These depend on the source language and will usually be more abstract than values at machine code level.

defined in the module (in-calls), (2) returns from such function calls (return-outs), (3) calls from within the module to a function defined outside the module (out-calls) and (4) returns from these calls (return-ins).

The semantics of a module M defines the valid traces of such interactions for M . For instance, the following traces are valid traces of the simple Counter module above:

- in-call `inc()`, return-out void, in-call `get_count()`, return-out 1
- in-call `inc()`, return-out void, in-call `inc()`, return-out void, in-call `get_count()`, return-out 2.

The following trace is a valid trace for the Counter module with an observer, where f_1 is an external function value:

- in-call `set_observer(f_1)`, return-out void, in-call `inc()`, out-call $f_1(1)$, return-in void, return-out void

The definition of a formal modular semantics for our simple C dialect should be a straightforward adaptation of the modular semantics for Java Jr. [JR05].

4.3 Defensive validation and interpretation of context-enclave interactions as source code interactions

Safety of an enclave requires well-defined reaction of the enclave to *any* **jump-in** action, but the modular source code semantics only defines the reaction to (well-typed) source-code level in-call or return-in actions. Therefore, the key to make a compiled enclave safe is to define the machine-code level *calling conventions*, and to defensively validate every **jump-in** action to make sure that under these calling conventions the **jump-in** action can be interpreted as a well-typed source code in-call or return-in.

For our simple language we define the following. A source code module will be compiled to an enclave that has an entry point e_f for each function f defined in the module, and one additional entry point e_{return} that we call the return entry point. The calling convention to call function f with arity n will be to jump to e_f passing arguments to f in registers 0 to $n - 1$, and the return address in register n . For simplicity, we assume that the processor has a sufficiently large number of registers. A return is performed by jumping from the module to the specified return address, with the result value in register 0.

Now (assuming a 32-bit processor), we define how to interpret machine code values (i.e. 32-bit words) as source code values of a specific expected type:

- Where the source module expects an integer value, the word is interpreted as a signed 32-bit integer.
- Where the source module expects void, any 32-bit word is interpreted as the value void.

- Where the source module expects a function pointer of a specific type, the word is interpreted as follows:
 - If the word is an address outside the enclave, the word is interpreted as an abstract external function value of the appropriate type.
 - If the word is an address inside the enclave, then it is considered valid only if it is equal to an entry point e_f for a function f defined in the module and of the appropriate type, and it is then interpreted as the internal function value f .

Next, we define how low-level context-enclave interactions relate to source code module interactions, making sure that *any* possible entry from the context to the enclave (that could hence possibly be malicious) either finishes execution with a run time error during validation, or can be related to a valid source code level module entry.

- A **jump-in** $e(v_i)$ interaction correspond to source code interactions as follows:

1. If e is the entry point e_f corresponding to function f with arity n , then v_0 up to v_{n-1} are interpreted as source code values s_0 to s_{n-1} as specified above using the signature of f to determine what source code types to expect. If some of these values cannot be given a valid interpretation, the **jump-in** interaction is interpreted as a run time error.

Next, the word v_n (the return address according to the calling conventions) is checked to be an address outside the enclave. If this check fails, the **jump-in** interaction is interpreted as a run time error. If the check succeeds, then:

- this **jump-in** interaction corresponds to an in-call of f with actual parameters s_i , and
 - v_n is used as the address to jump to on completion of this in-call. More precisely, when the source code semantics specifies that the return-out corresponding to this in-call should happen, the compiler will generate a **jump-out** event to address v_n , with the return value in the first register of the processor.
2. If e is the entry point e_{return} , then v_0 is interpreted as a source code value s_0 using the return type of the most recent out-call as the expected source code type. If this interpretation is valid, the interaction is interpreted as the return-in interaction with return value s_0 , otherwise the **jump-in** interaction is interpreted as a run time error.
- When the source code semantics specifies that an out-call interaction should happen, the compiler generates a **jump-out** $a(v_i)$ interaction where a is the external function value (which is an address outside the enclave as specified above), passing the values v_i in the processor registers as actual parameters, and the e_{return} entry point as return address.

Consider again the Counter with observer example. With the defensive checks specified informally above in place, the module is now safe: if the context provides an invalid function pointer within the module to the `set_observer()` function, this will lead to an immediate run time error, and hence launching a return-oriented-programming style attack against the module is now also ruled out. Of course, the context can still provide invalid *external* function pointers, but these can only lead to undefined behavior in the context, never to undefined behavior in the module.

4.4 Proving safety

Formalization of the problem, and proofs are work-in-progress. The steps required are:

1. Formalization of the source language. We should define a modular semantics for the source, possibly also showing that this modular semantics is *compatible* with pre-existing definitions of a whole program semantics.
2. Formalization of the algorithm for defensive validation of **jump-in** events (informally described in Section 4.3).
3. A proof that the combination of this algorithm with the modular source code semantics defines a safe enclave specification.

We are confident that the proofs work out for the case of the simple language we considered in this paper, but significant challenges remain to address more advanced language features. We intend to gradually extend the simple source language with language features such as bounds-checked arrays, objects and multi-threading, as well as with typing features such as ownership types.

5 Related Work

The work reported on in this paper is related to existing work on secure (fully abstract) compilation, and work on language interoperability.

Fully abstract compilation [Aba99] studies the problem of compiling from source languages to target languages (including, e.g. machine code) such that, roughly speaking, attackers at the target level have no more power than attackers at the source level. More formally, fully abstract compilation preserves and reflects contextual equivalence. Fully abstract compilation has recently been studied intensively [ASJP12, BA15, PAS⁺15, DPP16, JHA⁺15], including compilation towards protected module architectures. Compared to our approach, a fully abstract compiler provides strictly stronger guarantees. For instance, confidentiality properties (like noninterference properties) are preserved by fully abstract compilation, but not by our safe compiler. Yet, full abstraction is much harder to achieve, and is a property of the entire compiler, whereas safety can be proven without formalizing a full compiler. We also conjecture that safe

compilers can be more efficient and would incur less runtime overhead on the generated code than fully abstract compilers.

Work on language interoperability, foreign function interfaces or contracts [MF07, FF02, AJP15, Ahm15] studies interoperability between different languages, but in most work the emphasis is either (1) on interoperability between two safe languages where one is untyped and one is typed, and where contracts check the interactions between the untyped and typed world, or (2) on interoperability (“plumbing”) between a safe language and a trusted unsafe language that implements a native API for the safe language.

Finally, an important additional motivation for studying enclave safety is the support for state continuity that researchers are developing [PLD⁺11, SJP14]. A state-continuous enclave is protected from so-called rollback attacks that revert the enclave’s internal state to a stale version of that state, but at the same time state-continuous enclaves are enforced to process all their inputs and this can be problematic if an input is provided that makes the enclave crash. Safety of enclaves provides stronger assurance that the enclave implements sufficiently defensive input checking.

6 Conclusion

Protected module architectures open the possibility to execute software modules in enclaves and rely on a very small trusted computing base for their correct execution. However, these architectures do not protect against vulnerabilities in the modules themselves. We have reported on work in progress towards guaranteeing *safety* of protected enclaves, in the sense that the behavior of the enclave at machine code level is fully determined by the source code for the enclave. Although our approach provides weaker security guarantees than fully abstract compilation, we expect that safe enclaves would execute more efficiently. This enables the use of safe languages together with protected module architectures in domains where the performance overhead would otherwise be prohibitive and where the stronger guarantees provided by fully abstract compilation are not required. Use cases in the fields of embedded computing, ubiquitous computing, or the Internet of Things can benefit from safe enclaves.

Acknowledgments

This research is partially funded by project grants from the Research Fund KU Leuven, and from the Research Foundation Flanders (FWO).

References

- [Aba99] Martín Abadi. Protection in programming-language translations. In *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, pages 19–34, 1999.

- [Ahm15] Amal Ahmed. Verified compilers for a multi-language world. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, pages 15–31, 2015.
- [AJP15] Pieter Agten, Bart Jacobs, and Frank Piessens. Sound modular verification of C code executing in an unverified context. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15*, pages 581–594, New York, NY, USA, 2015. ACM.
- [ASJP12] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. Secure compilation to modern processors. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 171–185, 2012.
- [BA15] William J. Bowman and Amal Ahmed. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015*, pages 101–113, New York, NY, USA, 2015. ACM.
- [CBB⁺01] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. Formatguard: automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th conference on USENIX Security Symposium, SSYS'01*, pages 1–9, Berkeley, CA, USA, 2001. USENIX Association.
- [CDD⁺10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS'10*, pages 559–572, New York, NY, USA, 2010. ACM.
- [DPP16] Dominique Devriese, Marco Patrignani, and Frank Piessens. Fully-abstract compilation by approximate back-translation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*, pages 164–177, 2016.
- [EDFPT12] K. El Defrawy, A. Francillon, D. Perito, and G. Tsudik. Smart: Secure and minimal architecture for (establishing a dynamic) root of trust. In *Proceedings of the Network & Distributed System Security Symposium (NDSS), San Diego, CA, 2012*.
- [EYP10] Úlfar Erlingsson, Yves Younan, and Frank Piessens. Low-level software security by example. In *Handbook of Information and Communication Security*. Springer, 2010.

- [FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. *SIGPLAN Not.*, 37(9):48–59, September 2002.
- [Huq15] Numaan Huq. PoS RAM scraper malware: Past, present, and future. Technical report, Trend Micro, 2015.
- [Int14] Intel. *Intel Software Guard Extensions Programming Reference*. 2014.
- [JHA⁺15] Yannis Juglaret, Cătălin Hrițcu, Arthur Azevedo de Amorim, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Towards a fully abstract compiler using Micro-Policies: Secure compilation for mutually distrustful components. Technical Report, arXiv:1510.00697, 2015.
- [JMG⁺02] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference, ATEC '02*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.
- [JR05] Alan Jeffrey and Julian Rathke. Java Jr.: Fully abstract trace semantics for a core Java language. In *Proceedings of the 14th European Conference on Programming Languages and Systems, ESOP'05*, pages 423–438, Berlin, Heidelberg, 2005. Springer-Verlag.
- [KSSV14] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14*, pages 10:1–10:14, New York, NY, USA, 2014. ACM.
- [MAB⁺13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP'13*, page 8, New York, NY, USA, 2013. ACM.
- [MF07] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '07*, pages 3–10, New York, NY, USA, 2007. ACM.
- [MK14] Nicholas D. Matsakis and Felix S. Klock, II. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology, HILT '14*, New York, NY, USA, 2014. ACM.

- [MLQ⁺10] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.
- [MPP⁺08] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, pages 315–328. ACM, April 2008.
- [NAD⁺13] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 479–498, Washington, D.C., 2013. USENIX.
- [PAS⁺15] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure compilation to protected module architectures. *ACM Trans. Program. Lang. Syst.*, 37(2):6:1–6:50, April 2015.
- [Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [PLD⁺11] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.
- [Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security, CCS '07*, pages 552–561, New York, NY, USA, 2007. ACM.
- [SJP14] Raoul Strackx, Bart Jacobs, and Frank Piessens. ICE: A passive, high-speed, state-continuity scheme. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC '14*. ACM, 2014.
- [SP12] Raoul Strackx and Frank Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12*, pages 2–13, New York, NY, USA, 2012. ACM.

- [SPP10] Raoul Strackx, Frank Piessens, and Bart Preneel. Efficient isolation of trusted subsystems in embedded systems. In Sushil Jajodia and Jianying Zhou, editors, *SecureComm*, volume 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 344–361. Springer, 2010.
- [SYP⁺09] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, EuroSec’09, pages 1–8, New York, NY, USA, 2009. ACM.

Verification of privacy-type properties for security protocols with XOR

David Baelde*, Stéphanie Delaune*, Ivan Gazeau*, and Steve Kremer†

*LSV, ENS Cachan & CNRS, Université Paris-Saclay

†LORIA, INRIA Nancy - Grand-Est

Cryptographic protocols are distributed programs relying on cryptographic primitives, whose purpose is to provide various security guarantees. Some of these properties, *e.g.*, anonymity or unlinkability, can be expressed by means of protocol equivalence. Our main contribution is to extend AKISS, an automatic prover for equivalence properties, to enable it to deal with protocols using bit-wise XOR.

I. AKISS IN A NUTSHELL

Our work is an extension of an existing tool, AKISS [1], which verifies equivalence for bounded sessions of protocols. Precisely, AKISS considers processes without replication expressed in a restricted variant of the applied pi calculus. The definition of equivalence proved by AKISS is close to trace equivalence, and coincides with it when processes are determinate. The procedure to check equivalence is based on a fully abstract modeling of symbolic traces in first order Horn clauses. The primitives allowed in AKISS are the ones which can be expressed through a rewrite system that is convergent and enjoys the finite variant property. These conditions hold for most of cryptographic primitive like encryption, hashes, signature. However, the present version of AKISS does not deal with the bit-wise XOR primitive which is often used especially in RFID protocols.

The first step of the AKISS procedure is to turn traces of a process and attacker capabilities into Horn clauses. For instance, a trace

$$\mathbf{in}(x).\mathbf{out}(\mathbf{enc}(x, k))$$

where k is a secret key will be translated into

$$k(W_1, \mathbf{enc}(x, k)) \Leftarrow k(X, x) \quad (1)$$

which means that, if the attacker can derive x using the recipe (computation) X , then he can also obtain the knowledge of $\mathbf{enc}(x, k)$ as the first output of the channel (W_i is the i -th output). Precisely, to model which messages have been sent to the process and to maintain the chronology of the messages learned, a world is added to the k predicate. Therefore the complete Horn clause is

$$k_{[\mathbf{in}(x)]}(W_1, \mathbf{enc}(x, k)) \Leftarrow k_{\emptyset}(X, x)$$

Attacker capabilities are also described with Horn clauses. For instance, for the pair function, we have

$$k(\langle X, Y \rangle, \langle x, y \rangle) \Leftarrow k(X, x), k(Y, y).$$

This clause expresses that, if the attacker can derive x using recipe X and y using recipe Y , the compound recipe $\langle X, Y \rangle$ allows him to derive $\langle x, y \rangle$.

A key point in the Horn clause axiomatization used in AKISS is that it gets rid of the rewrite theory by considering variants. Variants avoid normalizing clause when doing substitution. For instance, a clause for the first projection would be

$$k(\pi_1(X), \pi_1(x)) \Leftarrow k(X, x).$$

However, it would produce terms not in normal form if x is substituted by $\langle u, v \rangle$. To avoid normalizing, AKISS pre-compute reductions, and these pre-computed reductions are called variants. In the case of the first projection, one of its variants is

$$k(\pi_1(X), x) \Leftarrow k(X, \langle x, y \rangle) \quad (2)$$

The equivalence property that AKISS is proving requires to find all identities of each trace. There is an identity when there are two different recipes which provide the same message (term). For instance the trace

$$\mathbf{out}(a).\mathbf{in}(x).\mathbf{out}(h(x))$$

allows to get $h(a)$ either by looking at the last output when a has been sent as an input or to directly compute $h(a)$ from a .

To find all these identities, AKISS computes a finite representation of all the knowledge (and all the recipes to obtain these knowledge) and then checks where are identities. This finite representation consists of a set of *solved* clauses. A clause is solved if all premisses of the clause contains terms which are just variables. For instance, (2) is not solved while (1) is.

To eliminate clauses which are not solved, AKISS composes them with all possible solved clauses to get new solved clauses, using a resolution rule which is a form of modus ponens between the conclusion of the solved clause and one premise of the unsolved clause. The formal definition is below where

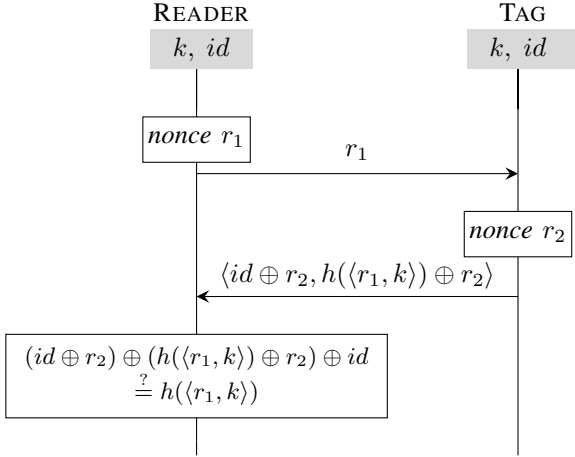
$$\begin{array}{l} f \in K, g \in K_{\text{solved}}, \\ f = \left(H \Leftarrow k_{uv}(X, t), B_1, \dots, B_n \right) \\ g = \left(k_w(R, t') \Leftarrow B_{n+1}, \dots, B_m \right) \\ \sigma = \text{mgu}(k_u(X, t), k_w(R, t')) \quad t \notin \mathcal{X} \\ \text{RESOLUTION} \quad \frac{}{K = K \uplus h \text{ where } h = \left((H \Leftarrow B_1, \dots, B_m) \sigma \right)} \end{array} \quad (3)$$

In the original tool, without the XOR primitive, this procedure has always terminated in all experiments.

II. ADDING XOR OPERATOR

A. Example

Example 1. Consider the RFID protocol depicted below:



The reader and the tag id share the secret key k . The reader starts by sending a nonce r_1 . The tag generates another nonce r_2 and computes the message $t_0 = \langle id \oplus r_2, h(\langle r_1, k \rangle) \oplus r_2 \rangle$. When receiving such a message, the reader will be able to retrieve r_2 , and by xoring it with the second component of the message he received, he obtains $h(\langle r_1, k \rangle)$.

Using our formalism, we can model the two roles of this protocol with the following ground linear processes:

$$P_{tag} = \mathbf{in}(c, x). \mathbf{out}(c, \langle id \oplus r_2, h(\langle x, k \rangle) \oplus r_2 \rangle). \mathbf{0}$$

$$P_{reader} = \mathbf{out}(c, r_1). \mathbf{in}(c, y).$$

$$[(proj_1(y) \oplus id) \oplus proj_2(y) \stackrel{?}{=} h(\langle r_1, k \rangle)]. \mathbf{0}$$

where $r_1, r_2, k \in \mathcal{N}$, $id \in \mathcal{N}_{pub}$, and $x, y \in \mathcal{X}$. The protocol itself corresponds to the set of ground linear processes obtained by interleaving these two roles.

B. Difficulty of the problem

The Xor operator $(\cdot + \cdot)$ cannot be treated as other primitive because the rewrite rules of the XOR, will not have the finite variant property (a term like $(\dots (a + x) + \dots) + a$ should be reduced after an arbitrary big number of permutation rules). To avoid this problem, we need, at least, to consider equality over terms modulo associativity and commutativity (AC). With such a change, unification of terms cannot be done exactly in the same way as before. Indeed, there is no most general unifier but a complete set of unifiers: a list which exhaustively represents all possible unifiers.

This modification is quite straightforward to implement, however without any additional change, the procedure will never terminate. Consider, for instance, the statement

$$k(W_2, c) \Leftarrow k(X, a + z). \quad (4)$$

This statement is not solved. Since we have the Horn clause corresponding to the constructor of the $+$:

$$k(X + Y, x + y) \Leftarrow k(X, x), k(Y, y), \quad (f_0^+)$$

we can do a resolution between the two clauses. The resolution requires that we unify $x + y$ with $a + z$. In our case the complete set of unifiers is the following:

$$x \mapsto a, y \mapsto z', z \mapsto z'$$

$$x \mapsto a + z_1, y \mapsto z_2, z \mapsto z_1 + z_2$$

$$x \mapsto a + z', y \mapsto 0, z \mapsto z'$$

and the other where x and y are swapped. While the first unifier leads to a solved statement :

$$k(W_2, c) \Leftarrow k(X, z'),$$

the second one produces the statement

$$k(W_2, c) \Leftarrow k(X, a + z_1), k(Y, z_2)$$

from which it is still possible to do a resolution against

$$k(X + Y, x + y) \Leftarrow k(X, x), k(Y, y)$$

again. Therefore the process will loop and the saturation will be never reached.

C. Our approach

To avoid non termination we add constraints on the resolution used by AKISS. For that, we introduce marking on atoms. A marked atom is an atom which cannot be unified with the head of the f_0^+ statement to achieve a RESOLUTION. Statements with marked literal are written $(H \Leftarrow B_1, \dots, B_n \parallel \mathcal{M})$ where \mathcal{M} is the subset of $\{B_1, \dots, B_n\}$ of marked atoms. Now, when a statement comes from the resolution of an unsolved statement with f_0^+ , we mark one of the two new atoms (see Figure 1). This atom is chosen such that it does not contain only variables: to select it we define $\text{rigid}(t)$, a function such that given a term $t = \sum_i t_i$ select one t_i which is not a variable. For instance the resolution of (4) with f_0^+ on the problematic unifier results in the following statement:

$$k(W_2, c) \Leftarrow k(X, a + z_1), k(Y, z_2) \parallel k(X, a + z_1)$$

note that $k(Y, z_2)$ may not have been marked since z_2 is a variable. The marking prevents resolution against (f_0^+) , therefore avoiding the diverging behavior described above.

The previous constraint is not enough by itself however. Indeed, it is inefficient when the term is $k(X, x + y)$: the two new terms are for one of the unifiers $(x \mapsto z_1 + z_3, y \mapsto z_2 + z_4)$ is $k(Y, z_1 + z_2), k(Z, z_3 + z_4)$. Therefore, whatever is the term we decide to mark, a resolution against f_0^+ is still possible on the other atom. This problem is particularly critical since among the variants of the clause f_0^+ , there is

$$k(X + Y, x + y) \Leftarrow k(X, x + u), k(Y, y + u),$$

which has the bad shape for premisses. To solve this problem, the procedure never does a resolution between this clause and f_0^+ .

These restrictions are the main ones, there is also a similar restriction on another rule of AKISS: EQUATION which prevent to test equality between the head of two statements if one of them is f_0^+ and other ones which are more technical like removing redundant premisses.

$$\begin{array}{c}
\text{RESOLUTION} \frac{
\begin{array}{l}
f = (H \Leftarrow k_{uv}(X, t), B_1, \dots, B_n \parallel \mathcal{M}) \in K \text{ such that } k_{uv}(X, t) \in \text{sel}(f) \\
g = (k_w(R, t') \Leftarrow B_{n+1}, \dots, B_m) \in K_{\text{solved}} \setminus f_0^+ \\
K = K \uplus \{h\sigma \mid \sigma \in \text{csu}_{\text{AC}}(k_u(X, t), k_w(R, t'))\} \\
\text{where } h = (H \Leftarrow B_1, \dots, B_n, B_{n+1}, \dots, B_m \parallel \mathcal{M} \setminus \{k_{uv}(X, t)\})
\end{array}
}{
\begin{array}{l}
f = (H \Leftarrow k_{uv}(X, t), B_1, \dots, B_n \parallel \mathcal{M}) \in K \text{ such that } k_{uv}(X, t) \in (\text{sel}(f) \setminus \mathcal{M}) \text{ and } a \in \text{rigid}(t) \\
g = (k_w(X_1 \oplus X_2, x_1 \oplus x_2) \Leftarrow k(X_1, x_1), k(X_2, x_2)) \in f_0^+ \\
K = K \uplus \{h\sigma \mid \sigma \in \text{csu}_{\text{AC}}(k_u(X, t), k_w(X_1 \oplus X_2, x_1 \oplus x_2)) \text{ and } a\sigma \in \text{factor}(x_1\sigma) \cup \{\perp\}\} \\
\text{where } h = (H \Leftarrow B_1, \dots, B_n, k_w(X_1, x_1), k_w(X_2, x_2) \parallel \mathcal{M} \cup \{k_w(X_1, x_1)\})
\end{array}
} \\
\text{RESOLUTION+}
\end{array}$$

Fig. 1: Saturation rules to replace (3)

Finally, we prove that the saturated set of clauses is correct, *i.e.*, it allows to derive enough knowledge to test trace equivalence. The rough idea of the proof is that due to equalities modulo AC, we can discard some statements as long as we ensure there are other statements where the recipes are equal to the formers modulo AC are present in the saturated set. The constraints we put on the rules enforce the recipes which contain Xor to have a sort of canonical form. This “canonical form” enforces that only the inner parenthesizing can contains recipes which produce terms with a shared factor and that outer parenthesizing has a list structure instead of a tree structure. For instance, if the recipe R_1 produces the term $a \oplus b$, R_2 produces the term $b \oplus c$, R_3 : d , and there is an identity which involves $R_1 \oplus R_2 \oplus R_3$ our constraint system will find it but with the parenthesizing $(R_1 \oplus R_2) \oplus R_3$ and not with the parenthesizing $(R_1 \oplus R_3) \oplus R_2$ because in the second case R_1 and R_3 produces terms which does not share any factor while $(R_1 \oplus R_3)$ and R_2 shares b .

III. BENCHMARK

We now report on experimental results. Sources and instructions for reproduction are available at <https://github.com/akiss/akiss/tree/xor>.

Our experiments include 5 RFID protocols using the exclusive-or operator from van Deursen and Radomirovic’s survey paper [2]. The KCL protocol corresponds to our running example (Example 1). For these protocols we have checked unlinkability, modelled as explained in Example 1, *i.e.*, we directly consider the linear process corresponding to two successive executions of a same or different tag. On 4 of the 5 protocols we find (known) attacks which violate unlinkability. Note that for our tool finding attacks or proving their absence are mainly equally difficult tasks, as for both we need to complete the saturation of the traces.

We have also encoded authentication properties as equivalences for the LAK protocol (on which unlinkability holds) and on a variant of the NSL protocol which uses xor [3]. We are able to find attacks on our authentication property on the LAK protocol and on the xor-based NSL variant.

The results are summarised in Figure 2. The models considered in these examples are generally simplified, and we

Protocol	Result	Protocol	Result
KCL	attack	LAK	safe
LD	attack	OTYT	attack
LAK	safe	YPL	attack
OTYT	attack	NSL xor	attack
YPL	attack		

(a) Verification of unlinkability (b) Verification of authentication

Fig. 2: Summary of case studies

only consider one interleaving modelled by a linear process as the aim of the examples is to illustrate that our resolution strategy allows the saturation procedure to terminate on existing protocols.

REFERENCES

- [1] R. Chadha, Ștefan Ciobăcă, and S. Kremer, “Automated verification of equivalence properties of cryptographic protocols,” in *Proc. 21st European Symposium on Programming (ESOP’12)*, ser. Lecture Notes in Computer Science, H. Seidl, Ed., vol. 7211. Springer, 2012, pp. 108–127.
- [2] T. van Deursen and S. Radomirovic, “Attacks on rfid protocols,” *IACR Cryptology ePrint Archive*, vol. 2008, p. 310, 2008.
- [3] Y. Chevalier, R. Küsters, M. Rusinowitch, and M. Turuani, “An NP decision procedure for protocol insecurity with XOR,” in *18th IEEE Symposium on Logic in Computer Science (LICS’03)*. IEEE Comp. Soc. Press, 2003.

Principles of Layered Attestation

Paul D. Rowe
prowe@mitre.org

The MITRE Corporation

Abstract. Systems designed with measurement and attestation in mind are often layered, with the lower layers measuring the layers above them. Attestations of such systems, which we call *layered attestations*, must bundle together the results of a diverse set of application-specific measurements of various parts of the system. Some methods of layered attestation are more trustworthy than others, so it is important for system designers to understand the trust consequences of different system configurations. This paper presents a formal framework for reasoning about layered attestations, and provides generic reusable principles for achieving trustworthy results.

1 Introduction

Security decisions often rely on trust. Many computing architectures have been designed to help establish the trustworthiness of a system through remote attestation. They gather evidence of the integrity of a target system and report it to a remote party who appraises the evidence as part of a security decision. A simple example is a network gateway that requests evidence that a target system has recently run antivirus software before granting it access to a network. If the virus scan indicates a potential infection, or does not offer recent evidence, the gateway might decide to deny access, or perhaps divert the system to a remediation network. Of course the antivirus software itself is part of the target system, and the gateway may require integrity evidence for the antivirus for its own security decision. This leads to the design of layered systems in which deeper layers are responsible for generating integrity evidence of the layers above them.

A simple example of a layered system is one that supports “trusted boot” in which a chain of boot-time integrity evidence is generated for a trusted computing base that supports the upper layers of the system. A more complex example might be a virtualized cloud architecture. The virtual machines (VMs) at the top are supported at a lower layer by a hypervisor or virtual machine monitor. Such an architecture may be augmented with additional VMs at an intermediate layer that are responsible for measuring the main VMs to generate integrity evidence. These designs offer exciting possibilities for remote attestation. They allow for specialization and diversity of the components involved, tailoring the capabilities of measurers to their targets of measurement, and composing them in novel ways.

However, the resulting layered attestations are typically more complex and challenging to analyze. Given a target system, what set of evidence should an appraiser request? What extra guarantees are provided if it receives integrity evidence of the measurers themselves? Does the order in which the measurements are taken matter? Can the appraiser tell if the correct sequence of measurements was taken?

This paper begins to tame the complexity surrounding attestations of these layered systems. We provide a formal model of layered measurement and attestation systems that abstracts away the underlying details of the measurements and focuses on the causal relationships among component corruption, measurement, and reporting. The model allows us to provide and justify generic, reusable strategies both for measuring system components and reporting the resulting integrity evidence.

Limitations of measurement. Our starting point for this paper is the recognition of the fact that measurement cannot *prevent* corruption; at best, measurement only *detects* corruption. In particular, the runtime corruption of a component can occur even if it is launched in a known good state. An appraiser must therefore always be wary of the gap between the time a component is measured and the time at which a trust decision is made. If the gap is large then so is the risk of a time-of-check-to-time-of-use (TOCTOU) attack in which an adversary corrupts a component during the critical time window to undermine the trust decision. A successful measurement strategy will limit the risk of TOCTOU attacks by ensuring the time between a measurement and a security decision is sufficiently small. The appraiser can then conclude that if the measured component is currently corrupted, it must be because the adversary performed a *recent* attack.

Shortening the time between measurement and security decision, however, is effective only if the measurement component can be trusted. By corrupting the measurer, an adversary can lie about the results of measurement making a corrupted target component appear to be in a good state. This affords the adversary a much larger window of opportunity to corrupt the target. The corruption no longer has to take place in the small window between measurement and security decision because the target can already be corrupted at the time of (purported) measurement. However, in a typical layered system design, deeper components such as a measurer have greater protections making it harder for an adversary to corrupt them. This suggests that to escape the burden performing a recent corruption, an adversary should have to pay the price of corrupting a *deep* component.

Formal model of measurement and attestation. With this in mind, our first main contribution is a formal model designed to aid in reasoning about what an adversary must do in order to defeat a measurement and attestation strategy. Rather than forbid the adversary from performing TOCTOU attacks in small windows or from corrupting deep components, we consider an attestation to be successful if the only way for the adversary to defeat its goals is to

perform such difficult tasks. Thus our model accounts for the possibility that an adversary might corrupt (and repair) arbitrary system components at any time. The model also features a true concurrency execution semantics which allows us to reason more directly about the causal effects of corruptions on the outcomes of measurement without having to reason about unnecessary interleavings of events. It has an added benefit of admitting a natural, graphical representation that helps an analyst quickly understand the causal relationships between events of an execution.

We demonstrate the utility of this formal model by validating the effectiveness of two strategies, one for the order in which to take measurements, the other for how to report the results in quotes from Trusted Platform Modules (TPMs). TPM is not the only technology available that provides a hardware root of trust for reporting. Indeed solutions may be conceived that use other external hardware security modules or emerging hardware support for trusted execution environments such as Intel’s SGX. However, most of the research on attestation is based on using a TPM as the hardware root of trust for reporting, and in this work, we follow that trend. We formally prove that under some assumptions about measurement and the behavior of uncorrupted components, in order for the adversary to defeat an attestation, he must perform some corruption which is “difficult.” The result is relatively concrete advice that can be applied by those building and configuring attestation systems. By implementing our general strategies and assumptions, layered systems can engage in more trustworthy attestations than might otherwise result.

Strategy for measurement. An intuition manifest in much of the literature on measurement and attestation is that trust in a system should be based on a bottom-up chain of measurements starting with a hardware root of trust for measurement. This is the core idea behind trusted boot processes, in which one component in the boot sequence measures the next component before launching it. Theorem 1, which we refer to as the “recent or deep” theorem, validates this common intuition and solidifies exactly how an adversary can defeat such bottom-up measurement strategies. It roughly says the following:

If a system has measured deeper components before more shallow ones, then the only way for the adversary to corrupt a component t without detection is either by *recently* corrupting one of t ’s dependencies, or else by corrupting a component even *deeper* in the system.

Strategy for bundling evidence. Given the importance of the order of measurement, it is also important for an attestation to reliably convey not only the outcome of measurements, but the order in which they were taken. This point is frequently overlooked in the literature on TPM-based attestation. Unfortunately, the structure of TPM quotes does not always reflect this ordering information, especially if some of the components depositing measurement values might be dynamically corrupted. We thus propose a particular strategy for creating a bundle of evidence in TPM quotes designed to give evidence that measurements were

indeed taken bottom up. We show in Theorem 2 that, under certain assumptions about the uncorrupted measurers in the system, this strategy preserves the guarantees of bottom-up measurement in the following sense:

If the system satisfies certain assumptions, and the TPM quote formed according to our bundling strategy indicates no corruptions, then either the measurement were really taken bottom-up, or the adversary *recently* corrupted one of t 's dependencies, or else the adversary corrupted an even *deeper* component.

Thus, any attempt the adversary makes to avoid the conditions for the hypothesis of Theorem 1 force him to validate its conclusion nonetheless.

Paper structure. The rest of the paper provides motivating examples and only a selection of the relevant formal details. We direct the reader to the full paper [13] for further details. Section 2 presents some relevant, related work. We motivate our intuitions and informally introduce our model in Section 3. In Section 4 we provide enough technical details about the relevant definitions to formally state the consequences of applying the intuition that it is better to measure “bottom-up.” In Section 5, provide examples of how TPMs can be misused, not providing the guarantees one might expect. We extend our model with more definitions in Section 6, providing just enough details to state the guarantees provided by a particular strategy for using TPMs to bundle evidence. We conclude in Section 7 pointing to directions for future work.

2 Related work.

There has been much research into measurement and attestation. While a complete survey is infeasible for this paper, we mention the most relevant highlights in order to describe how the present work fits into the larger context of research in this area. We divide the work into several broad categories. Although the boundaries between the categories can be quite blurry, we believe it helps to structure the various approaches.

Measurement techniques. Much of the early work was focused on techniques for measuring low-level components that make up a trusted computing base (TCB). These ideas have matured into implementations such as Trusted Boot [11]. Recognizing that many security failures cannot be traced back to the TCB, Sailer et al. [14] proposed an integrity measurement architecture (IMA) in which each application is measured (by hashing its code) before it is launched. More recently, there has been work trying to identify and measure dynamic properties of system components in order to create a more comprehensive picture of the runtime state of a system [10,9,5,15]. All these efforts try to establish what evidence is useful for inferring system state relevant to security decisions. The present work takes for granted that such special purpose measurements can be taken and that they will accurately reflect the system state. Rather, our focus is on developing principles for how to combine a variety of these measurers in

a layered attestation. We envision a system designer choosing the measurement capabilities that best suit her needs and using our work to ensure an appraiser can trust the integrity of the result.

Modular attestation frameworks. Cabuk and others [1] have proposed an architecture designed to support layered platforms with hierarchical dependencies. Their design introduces trusted software into the TCB as a software-based root of trust for measurement (SRTM). Although they explain how measurements by the SRTM integrate with the chain of measurements stored in a TPM, they do not study the effect corruptions of various components have on the outcome of attestations. In [2], Coker et al. identify five guiding principles for designing an architecture to support remote attestation. They also describe the design of a (layered) virtualized system based on these principles, although there does not appear to be a publicly available implementation at the time of writing. Of particular interest is a section that describes a component responsible for managing attestations. The emphasis is on the mechanics of selecting measurement agents by matching the evidence they can generate to the evidence requested by an appraiser. There is no discussion or advice regarding the relative order of measurements or the creation of an evidence bundle to reflect the order. More recently, modular attestation frameworks instantiating [2]’s principles have been implemented [8,7,3]. These are integrated frameworks that offer plug-and-play capabilities for measurement and attestation for specific usage scenarios. It is precisely these types of systems (in implementation or design) to which our analysis techniques would be most useful. We have not been able to find a discussion of the potential pitfalls of misconfiguring these complex systems. Our work should be able to help guide the configuration of such systems and analyze particular attestation scenarios for each architecture.

Attestation Protocols. Finally, works such as [2,6,4,12] study the properties of attestation protocols, typically protocols that use a TPM to report on integrity evidence provided by measurement agents. They tend to focus on the cryptographic protections required to secure the evidence as it is sent over a network. [2] proposes a protocol that binds the evidence to a session key, so that an appraiser can be guaranteed that subsequent communications will occur with the appraised system, and not a corrupted substitute. [6] and [12] examine the ways in which cryptographic protections for network events interact with the long-term state of a TPM. None of these consider the measurement activities on the target platform itself and how corruptions of components can affect the outcome of the protocol. In [4], Datta et al. introduce a formalism that accounts for actions local to the target machine as well as network events such as sending and receiving messages. Although they give a very careful treatment of the effect of a corrupted component on an attestation, their work differs in two key ways. First, the formalism represents many low-level details making their proof rather complex, sometimes obscuring the underlying principles. Second, their framework only accounts for static corruptions, while ours is specifically designed around the possibility of dynamic corruption and repair of system components.

3 Motivating Examples of Measurement

Consider an enterprise that would like to ensure that systems connecting to its network first demonstrate evidence of being in a secure state. Such evidence could take many forms from advanced measurements tailored to specific use cases, to the results of a simple virus scan. Indeed, layered architectures have a unique ability to support diversity in how such evidence is gathered. For the purposes of simplifying the exposition, we assume the gateway requires the results of a fresh system scan by the most up-to-date virus checker. The network gateway should ask systems to perform a system scan on demand when they attempt to connect. We may suppose the systems all have some component A_1 that is capable of accurately reporting the running version of the virus checker. Because this enterprise values high assurance, the systems also come equipped with another component A_2 capable of measuring the runtime state of the kernel. This is designed to detect any rootkits that might try to undermine the virus checker's system scan. We may assume that A_1 and A_2 are both measured by a root of trust for measurement (rtm) as part of a secure boot process.

We are thus interested in a system consisting of the following components: $\{\text{sys}, \text{vc}, \text{ker}, A_1, A_2, \text{rtm}\}$, where sys represents the collective parts of the system scanned by the virus checker vc , and ker represents the kernel. Based on the scenario described above, we may be interested in the following set of measurement events

$$\{\text{ms}(\text{rtm}, A_1), \text{ms}(\text{rtm}, A_2), \text{ms}(A_1, \text{vc}), \text{ms}(A_2, \text{ker}), \text{ms}(\text{vc}, \text{sys})\}$$

where $\text{ms}(o_1, o_2)$ represents the measurement of o_2 by o_1 . These measurement events generate the raw evidence that the network gateway can use to make a determination as to whether or not to admit the system to the network.

If any of the measurements indicate a problem, such as a failed system scan, then the gateway has good reason to believe it should deny the system access to the network. But what if all the evidence it receives looks good? How confident can the gateway be that the version and signature files are indeed up to date? The answer will depend on the order in which the evidence was gathered. To get some intuition for why this is the case, consider the three different specifications pictured in Fig. 1 for how to order the measurements. (The bullet after the first three events is inserted only for visible legibility, to avoid crossing arrows.)

Specification S_1 ensures that both vc and ker are measured before vc runs its system scan. Specifications S_2 and S_3 each relax one of those ordering requirements. Let's now consider some executions that respect the order of measurements in each of these specifications in which the adversary manages to avoid detection.

Execution E_1 of Fig. 2 is compatible with Specification S_1 . The adversary manages to corrupt the system by installing some user-space malware sometime in the past. If we assume the up-to-date virus checker is capable of detecting this malware, then the adversary must corrupt either vc or ker before the virus scan represented by $\text{ms}(\text{vc}, \text{sys})$. That is, either a corrupted vc will lie about the

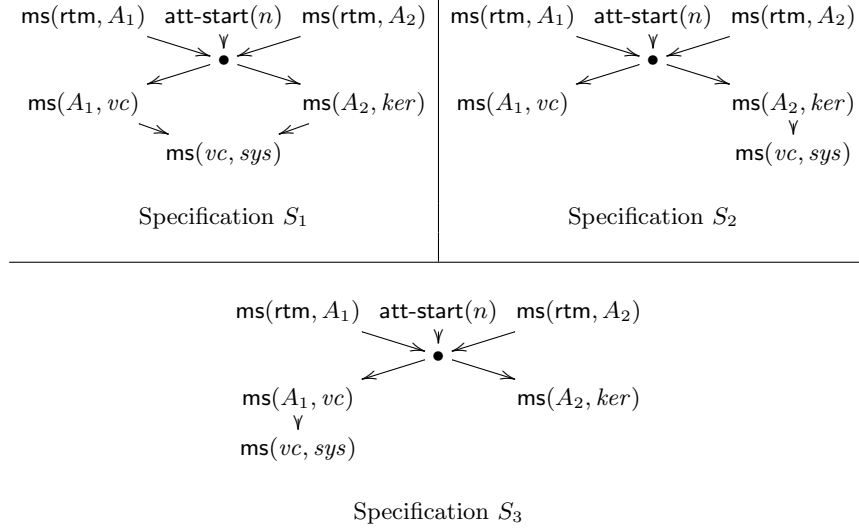


Fig. 1. Three orders for measurement

results of measurement, or else a corrupted ker can undermine the integrity of the system scan, for example, by hiding the directory containing the malware from vc . In the case of E_1 , the adversary corrupts vc in order to lie about the results of the system scan, but it does so after $ms(A_1, vc)$ in order to avoid detection by this measurement event.

In Execution E_2 , which is consistent with Specification S_2 , the adversary is capable of avoiding detection while corrupting vc much earlier. The system scan $ms(vc, sys)$ is again undermined by the corrupted vc . Since vc will also be measured by A_1 , the adversary has to restore vc to an acceptable state before $ms(A_1, vc)$. Execution E_3 is analagous to E_2 , but the adversary corrupts ker instead of vc , allowing it to convince the uncorrupted vc that the system has no malware. Since Specification S_3 allows $ms(A_1, vc)$ to occur after the system scan, the adversary can leverage the corrupted vc to lie about the scan results, but must restore vc to a good state before it is measured.

Execution E_1 is ostensibly harder to achieve for the adversary than either E_2 or E_3 , because the adversary has to work quickly to corrupt vc *during* the attestation. In E_2 and E_3 , the adversary can corrupt vc and ker respectively at any time in the past. He still must perform a quick restoration of the corrupted component during the attestation, but there are reasons to believe this may be easier than corrupting the component to begin with. Is it true that *all* executions respecting the measurement order of S_1 are harder to achieve than E_2 and E_3 ? What if the adversary corrupts vc before the start of the attestation? It would seem that he would also have to corrupt A_1 to avoid detection by A_1 's measurement of vc , $ms(A_1, vc)$.

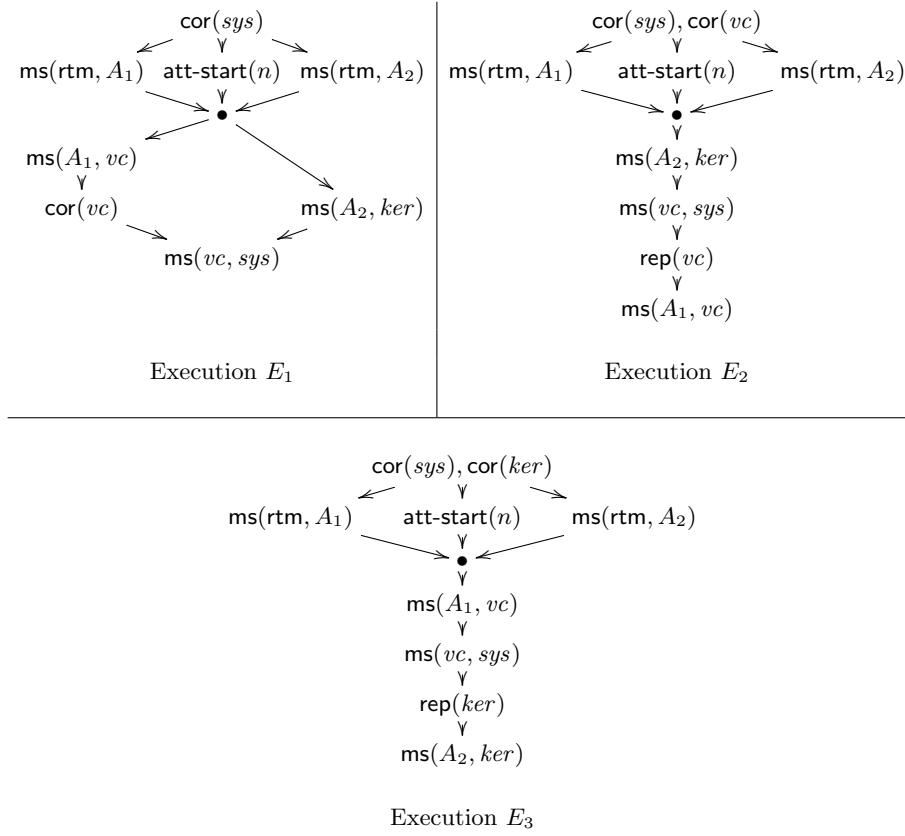


Fig. 2. Three system executions

One major contribution of this paper is to provide a formal framework in which to ask and answer such questions. Within this framework we can begin to characterize what the adversary must do in order to avoid detection by measurement. We will show that there is a precise sense in which Specification S_1 is strictly stronger than S_2 or S_3 . This is an immediate corollary of a more general result (Theorem 1) that validates a strong intuition that pervades much of the literature on measurement and attestation: Attestations are more trustworthy if the lower-level components of a system are measured before the higher-level components. The next section lays the groundwork for this result.

4 Measurement Systems

4.1 Preliminaries and Definitions

In this section we formalize the intuitions we used for the examples in the previous section. We start by defining measurement systems which perform the core functions of creating evidence for attestation.

System architecture.

Definition 1. We define a measurement system to be a tuple $\mathcal{MS} = (O, M, C)$, where O is a set of objects (e.g. software components) with a distinguished element rtm . M and C are binary relations on O . We call

M the measures relation, and
 C the context relation.

We say M is rooted when for every $o \in O \setminus \{\text{rtm}\}$, $M^+(\text{rtm}, o)$, where M^+ is the transitive closure of M .

We henceforth assume M is rooted and acyclic. As a consequence, rtm is not the target of any measurement, i.e. $\neg M(o, \text{rtm})$ for any $o \in O$. We also assume the context relation C is transitive and acyclic, so that no object relies on itself for its own clean runtime context. Given an object $o \in O$ we define the measurers of o to be $M^{-1}(o) = \{o' \mid M(o', o)\}$. We similarly define the context for o to be $C^{-1}(o)$. We extend these definitions to sets in the natural way.

We additionally assume $M \cup C$ is acyclic. This ensures that the combination of the two dependency types does not allow an object to depend on itself. Such systems are stratified, in the sense that we can define an increasing set of dependencies as follows.

$$\begin{aligned} D^1(o) &= M^{-1}(o) \cup C^{-1}(M^{-1}(o)) \\ D^{i+1}(o) &= D^1(D^i(o)) \end{aligned}$$

So $D^1(o)$ consists of the measurers of o and their context. As we will see later, $D^1(o)$ represents the set of components that must be uncompromised in order to trust the measurement of o .

We can represent measurement systems pictorially as a graph whose vertices are the objects of \mathcal{MS} and whose edges encode the M and C relations. We use the convention that $M(o_1, o_2)$ is represented by a solid arrow from o_1 to o_2 , while $C(o_1, o_2)$ is represented by a dotted arrow from o_1 to o_2 . The representation of the system described in Section 3 is shown in Figure 3.

Terms and derivability. It is called a measurement system because the primary activity of these components is to measure each other. The results of measurement are expressed using elements of a term algebra.

Terms are constructed from some base V of atomic terms using constructors in a signature Σ . The set of terms is denoted $\mathcal{T}_\Sigma(V)$. We assume Σ includes at least some basic constructors such as pairing (\cdot, \cdot) , signing $\llbracket (\cdot) \rrbracket_{(\cdot)}$, and hashing

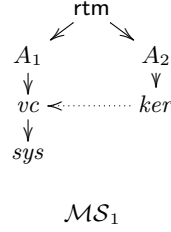


Fig. 3. Visual representation of an example measurement system.

$\#(\cdot)$. The set V is partitioned into public atoms \mathcal{P} , random nonces \mathcal{N} , and private keys \mathcal{K} .

Our analysis will sometimes depend on what terms an adversary can derive (or construct). We say that term t is derivable from a set of terms $T \subseteq V$ iff $t \in \mathcal{T}_\Sigma(T)$, and we write $T \vdash t$. We assume the adversary knows all the public atoms \mathcal{P} , and so can derive any term in $\mathcal{T}_\Sigma(\mathcal{P})$ at any time. For each $o \in O$, we assume there is a distinguished set of (public) measurement values $\mathcal{MV}(o) \subset \mathcal{P}$.

Events, outputs, and executions. The components $o \in O$ and the adversary on this system perform actions. In particular, objects can measure each other and the adversary can corrupt and repair components in an attempt to influence the outcome of future measurement actions. Additionally, an appraiser has the ability to inject a random nonce $n \in \mathcal{N}$ into an attestation in order to control the recency of events.

Definition 2 (Events). Let \mathcal{MS} be a target system. An event for \mathcal{MS} is a node e labeled by one of the following.

- A measurement event is labeled by $\text{ms}(o_2, o_1)$ such that $M(o_2, o_1)$. We say such an event measures o_1 , and we call o_1 the target of e . We let $\text{Supp}(e)$ denote the set $\{o_2\} \cup C^{-1}(o_2)$.
- An adversary event is labeled by either $\text{cor}(o)$ or $\text{rep}(o)$ for $o \in O \setminus \{\text{rtm}\}$.
- The attestation start event is labeled by $\text{att-start}(n)$, where n is a term.

When an event e is labeled by ℓ we will write $e = \ell$. We often refer to the label ℓ as an event when no confusion will arise.

An event e touches o , iff either

- o is an argument to the label of e , or
- $o \in \text{Supp}(e)$.

The $\text{att-start}(n)$ event will serve to bound events in time. It represents the random choice by the appraiser of the value n . The appraiser will know that anything occurring after this event can reasonably be said to occur “recently”. The measurement events are self explanatory. Adversary events represent the corruption ($\text{cor}(\cdot)$) and repair ($\text{rep}(\cdot)$) of components. Notice that we have excluded rtm from corruption and repair events. This is not because we assume the rtm

to be immune from corruption, but rather because all the trust in the system relies on the `rtm`: Since it roots all measurements, if it is corrupted, none of the measurements of other components can be trusted.

We organize the events of an execution by a partially ordered set of events (E, \prec) . When no confusion arises, we often refer to (E, \prec) by its underlying set E and use \prec_E for its order relation. Given a poset (E, \prec) , let $e\downarrow = \{e' \mid e' \prec e\}$, and $e\uparrow = \{e' \mid e \prec e'\}$. Given a set of events E , we denote the set of adversary events of E by $adv(E)$ and the set of measurement events by $meas(E)$.

In order to make sense of the outputs of measurements, we cannot consider arbitrary partially ordered sets of events. Since the output of a measurement depends on the corruption state of the target, the measurer, and the measurer's context, we must ensure that the corruption state of a component is well-defined. In the full version [13], we thus introduce a notion of a partially ordered sets of events being *adversary-ordered*, and we prove that this ensure the following notion of corruption state is well-defince.

Definition 3 (Corruption state). *Let (E, \prec) be a finite, adversary-ordered poset for \mathcal{MS} . For each event $e \in E$ and each object o the corruption state of o at e , written $cs(e, o)$, is an element of $\{\perp, r, c\}$ and is defined as follows. $cs(e, o) = \perp$ iff $e \notin E_o$. Otherwise, we define $cs(e, o)$ inductively:*

$$cs(e, o) = \begin{cases} c & : e = \text{cor}(o) \\ r & : e = \text{rep}(o) \\ r & : e \in meas(E) \wedge adv(e\downarrow) \cap E_o = \emptyset \\ cs(e', o) & : e \in meas(E) \wedge e' \text{ maximal in } adv(e\downarrow) \cap E_o \end{cases}$$

When $cs(e, o)$ takes the value c we say o is corrupt at e ; when it takes the value r we say o is uncorrupt or regular at e ; and when it takes the value \perp we say the corruption state is undefined.

We make a key simplifying assumption that measurements by uncorrupted components accurately reflect the corruption state of the target of measurement, and an appraiser can accurately identify which outputs reflect corruption.

Assumption 1 (Measurement Accuracy) *Let $\mathcal{G}(o)$ and $\mathcal{B}(o)$ be a partition for $\mathcal{MV}(o)$. Let $e = \text{ms}(o_2, o_1)$. The output of e , written $out(e)$, is defined as follows. $out(e) = v \in \mathcal{B}(o_1)$ iff $cs(e, o_1) = c$ and for every $o \in \{o_2\} \cup \{o' \mid C(o', o_2)\}$, $cs(e, o) = r$. Otherwise $out(e) = v \in \mathcal{G}(o_1)$.*

If $out(e) \in \mathcal{B}(o_1)$ we say e detects a corruption. If $out(e) \in \mathcal{G}(o_1)$ but $cs(e, o_1) = c$, we say the adversary avoids detection at e .

If $e = \text{att-start}(n)$, then $out(e) = n$.

Informally stated, this says that, as long as the components involved in taking a measurement are not corrupted, measurements produce evidence that allow an appraiser to make a determination of the corruption state of the target *without false positives or negatives*. While in reality false positives and negatives will occur for many reasons, this assumption allows us to focus on the logical structure

of attestations before confronting the consequences of uncertainty that arise due to measurement inaccuracies.

We can now define what it means to be an execution of a measurement system.

Definition 4 (Executions, Specifications). *Let \mathcal{MS} be a measurement system.*

1. *An execution of \mathcal{MS} is any finite, adversary-ordered poset E for \mathcal{MS} .*
2. *A specification for \mathcal{MS} is any execution that contains no adversary events.*

Specification S admits an execution E iff there is an injective, label-preserving map of partial orders $\alpha : S \rightarrow E$. The set of all executions admitted by S is denoted $\mathcal{E}(S)$.

Measurement specifications are the way an appraiser might ask for measurements to be taken in a particular order. The set $\mathcal{E}(S)$ is just the set of executions in which the given events have occurred in the desired order. The appraiser can thus analyze $\mathcal{E}(S)$ in advance to determine what an adversary has to do to avoid detection, given that the events in S were performed as specified.

The question of how an appraiser learns whether or not the actual execution performed is in $\mathcal{E}(S)$ is an important one. The second half of the paper is dedicated to that problem. For now, we consider what an appraiser can infer about an execution E given that $E \in \mathcal{E}(S)$.

4.2 A Strategy for Measurement

We now turn to a formalization of the rule of thumb at the end of Section 3. By ensuring that specifications have certain structural aspects, we can conclude the executions they admit satisfy useful constraints. In particular, it is useful to measure components from the bottom up with respect to the dependencies of the system. That is, if whenever o_1 depends on o_2 we measure o_2 before measuring o_1 , then we can usefully narrow the range of actions the adversary must take in order to avoid detection. For this discussion we fix a target system \mathcal{MS} . Recall that $D^1(o)$ represents the measurers of o and their runtime context.

Definition 5. *A measurement event $e = \text{ms}(o_2, o_1)$ in execution E is well-supported iff either*

- i. $o_2 = \text{rtm}$, or
- ii. *for every $o \in D^1(o_1)$, there is a measurement event $e' \prec_E e$ such that o is the target of e' .*

When e is well-supported, we call the set of e' from Condition ii above the support of e . An execution E measures bottom-up iff each measurement event $e \in E$ is well-supported.

Theorem 1 (Recent or deep). *Let E be an execution with well-supported measurement event $e = \text{ms}(o_1, o_t)$ where $o_1 \neq \text{rtm}$. Suppose that E detects no corruptions. If the adversary avoids detection at e , then either*

1. there exist $o \in D^1(o_t)$ and $o' \in M^{-1}(o)$ such that $\text{ms}(o', o) \prec_E \text{cor}(o) \prec_E e$
2. there exists $o \in D^2(o_t)$ such that $\text{cor}(o) \prec_E e$.

Proof. The proof can be found in [13].

This theorem says, roughly, that if measurements indicate things are good when they are not, then there must either be a recent corruption or a deep corruption. This tag line of “recent or deep” is particularly apt if the system dependencies also reflect the relative difficulty for an adversary to corrupt them. By ordering the measurements so that more robust ones are measured first, it means that for an adversary to avoid detection for an easy compromise, he must have compromised a measurer since it itself was measured, or else, he must have previously (though not necessarily recently) compromised a more robust component. In this way, the measurement of a component can raise the bar for the adversary. If, for example, a measurer sits in a privileged location outside of some VM containing a target, it means that the adversary would also have to break out of the target VM and compromise the measurer to avoid detection. The skills and time necessary to perform such an attack are much greater than simply compromising the end target.

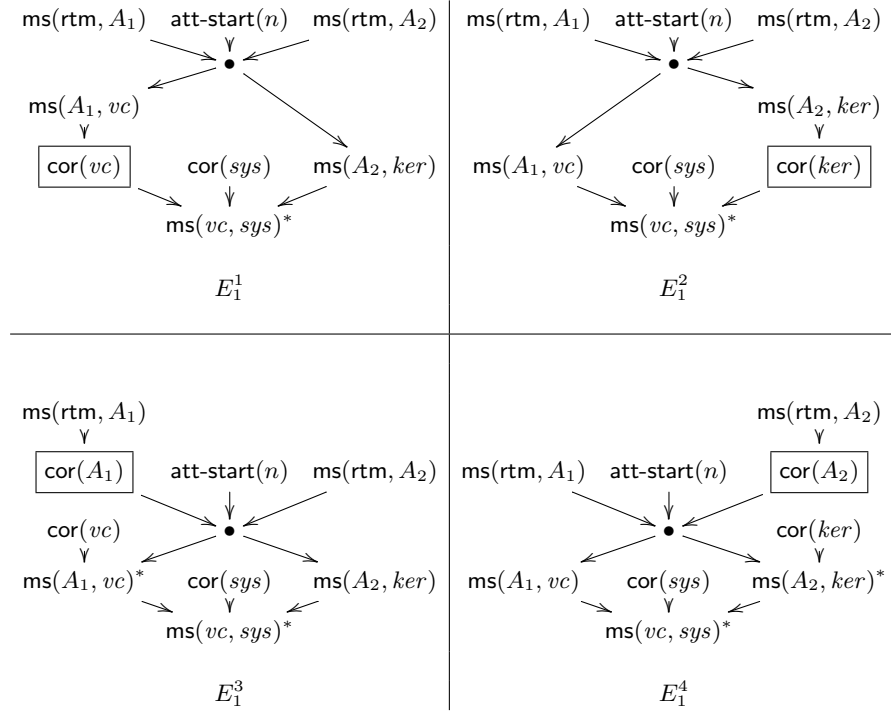


Fig. 4. Executions that do not detect corruption of sys .

Consider the bottom-up specification S_1 from Fig. 1. Theorem 1 indicates that if sys was corrupted at the time it was measured, then either vc or ker was recently corrupted, or else some deeper component was (not necessarily recently) corrupted. All the various options are shown in Figure 4 in which the measurement events at which the adversary avoids detection are marked with an asterisk, and the corruption events guaranteed by the theorem are boxed. Our theorem allows us to know that these executions essentially characterize all the cases in which a corrupted sys goes undetected.

5 Motivating Examples of Bundling

The previous section discusses how to constrain adversary behavior using the order of measurements. However, implicit in the analysis is the assumption that an appraiser is able to verify the order and outcome of the measurement events. Since a remote appraiser cannot directly observe the target system, this assumption must be discharged in some way. A measurement system must be augmented with the ability to record and report the outcome and order of measurement events. We refer to these additional activities as *bundling* evidence. Our focus for this paper is on using the Trusted Platform Module for this purpose. While there are techniques and technologies that can be used as roots of trust for reporting (e.g. hardware-based trusted execution environments such as Intel’s SGX) there has been a lot of research into TPMs and their use for attestation. Much of that work does not pay close attention to the importance of faithfully reporting the order in which measurements have taken place. Thus, we believe that studying TPM-based attestation is a fruitful place to start, and we leave investigations of other techniques and technologies for future work.

TPMs, PCRs, and quotes. In our presentation, we assume the reader has a basic familiarity with a few key features of Trusted Platform Modules (TPMs). In particular, we assume a familiarity with Platform Configuration Registers (PCRs) and how their contents can change through the *extend* command. We also assume familiarity with how TPM quotes work. More detailed background is provided in [13].

We represent both the values stored in PCRs and the quotes as terms in $\mathcal{T}_\Sigma(V)$. Since PCRs can only be updated by extending new values, their contents form a hash chain $\#(v_n, \#(\dots, \#(v_1, \text{rst})))$. We abbreviate such a hash chain as $\text{seq}(v_1, \dots, v_n)$. So for example, $\text{seq}(v_1, v_2) = \#(v_2, \#(v_1, \text{rst}))$. We say a hash chain $\text{seq}(v_1, \dots, v_n)$ *contains* v_i for each $i \leq n$. Thus the contents of a PCR contain exactly those values that have been extended into it. We also say v_i is *contained before* v_j in $\text{seq}(v_1, \dots, v_n)$ when $i < j \leq n$. That is, v_i is contained before v_j in the contents of p exactly when v_i was extended before v_j .

A quote from TPM t is a term of the form $\llbracket n, (p_i)_{i \in I}, (v_i)_{i \in I} \rrbracket_{sk(t)}$. It is a signature over a nonce n , a list of PCRs $(p_i)_{i \in I}$ and their respective contents $(v_i)_{i \in I}$ using $sk(t)$, the secret key of t . We always assume $sk(t) \in \mathcal{K}$ the set of non-public, atomic keys. That means the adversary does not know $sk(t)$ and hence cannot forge quotes.

5.1 Pitfalls of TPM-Based Bundling.

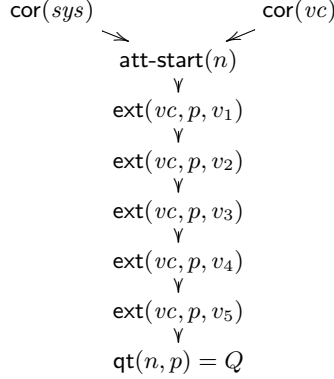
The two key features of TPMs (protected storage and secure reporting) allow components to store the results of their measurements and later report the results to a remote appraiser. The resulting quote (or set of quotes) is a bundle of evidence that the appraiser must use to evaluate the state of the target system. Indeed, this bundle is the only evidence the appraiser receives. In the rest of this section we present various examples that demonstrate how the structure of this bundle affects the trust inferences a remote appraiser is justified in making about the target.

Consider \mathcal{MS}_1 found in Section 3, and pictured in Fig. 3. Ideally a remote appraiser would be able to verify that an execution that produces a particular set of quotes \mathcal{Q} is in $\mathcal{E}(S_1)$ (from Fig. 1). The appraiser must be able to do this on the basis of \mathcal{Q} only. The possibilities for \mathcal{Q} depend somewhat on how \mathcal{MS}_1 is divided. For example, if \mathcal{MS}_1 is a virtualized system, *rtm* might sit in an administrative VM, and A_1 and A_2 could be in a privileged “helper” VM separated from the main VM that hosts *ker*, *vc*, and *sys*. If each of these VMs is supported by its own TPM, then \mathcal{Q} would have to contain at least three quotes just to convey the raw measurement evidence. However, if \mathcal{MS}_1 is not virtualized, they might all share the same TPM and a single quote might suffice. For our purposes it suffices to consider a simple architecture in which all the components share a single TPM.

Strategy 1: A single hash chain. Since PCRs contain an ordered history of the extended values, the first natural idea is for all the components to share a PCR p , each extending their measurements into p . The intuition is that the contents of p should represent the order in which the measurements occurred on the system. To make this more concrete, assume the measurement events of S_1 have the following output: $out(ms(rtm, A_1)) = v_1$, $out(ms(rtm, A_2)) = v_2$, $out(ms(A_1, vc)) = v_3$, $out(ms(A_2, ker)) = v_4$, $out(ms(vc, ker)) = v_5$. Then this strategy would produce a single quote $Q = \llbracket n, p, seq(v_1, v_2, v_3, v_4, v_5) \rrbracket_{sk(t)}$. To satisfy the order of S_1 , any linearization of the measurements would do, so the appraiser should also be willing to accept $Q' = \llbracket n, p, seq(v_2, v_1, v_3, v_4, v_5) \rrbracket_{sk(t)}$ in which v_1 and v_2 were generated in the reverse order.

Figure 5 depicts an execution that produces the expected quote Q , but does not satisfy the desired order. Since all the measurement components have access to the same PCR, if any of those components is corrupted, it can extend values to make it look as though other measurements were taken although they were not. This is particularly troublesome when a relatively exposed component like *vc* can impersonate the lower-level components that measure it.

This motivates our desire to have strict access control for PCRs. This would allow the appraiser to correctly infer which component has provided each piece of evidence. The locality feature of TPMs could be used for this purpose. Given the limitations of locality in the current technology, however, it may be necessary to introduce another component that is responsible for disambiguating the source of each measurement into a PCR. Such a strategy would require careful

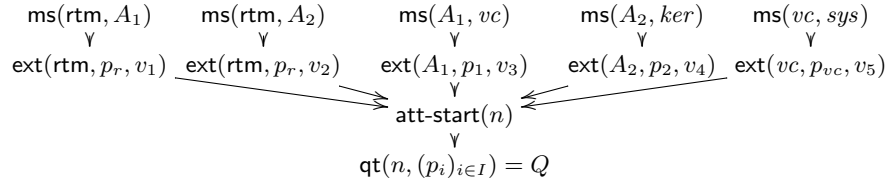


Output of quote is $Q = \llbracket n, p, \text{seq}(v_1, v_2, v_3, v_4, v_5) \rrbracket_{sk(t)}$.

Fig. 5. Defeating Strategy 1

consideration of the effect of a corruption of that component, and to include measurement evidence that it is functioning properly. For simplicity of our main analysis we freely take advantage of the assumption that TPMs can provide dedicated access to one PCR per component of the system it supports, leaving an analysis of the more complicated architecture for a more complete treatment of the subject.

Strategy 2: Separate hash chains. A natural next attempt given this assumption would be to produce a single quote over the set of PCRs that contain the measurement evidence. This would produce quotes with the structure $Q = \llbracket n, (p_r, p_1, p_2, p_{vc}), (s_1, s_2, s_3, s_4) \rrbracket_{sk(t)}$, in which $s_1 = \text{seq}(v_1, v_2)$, $s_2 = \text{seq}(v_3)$, $s_3 = \text{seq}(v_4)$, $s_4 = \text{seq}(v_5)$. Figure 6 demonstrates a failure of this strategy. The problem, of course, is that, since the PCRs may be extended concurrently, the relative order of events is not captured by the structure of the quote.



Output of quote is $Q = \llbracket n, (p_r, p_1, p_2, p_{vc}), (s_1, s_2, s_3, s_4) \rrbracket_{sk(t)}$
 $s_1 = \text{seq}(v_1, v_2), s_2 = \text{seq}(v_3), s_3 = \text{seq}(v_4), s_4 = \text{seq}(v_5)$.

Fig. 6. Defeating Strategy 2

Strategy 3: Tiered, nested quotes. We thus require a way to re-introduce evidence about the order of events while maintaining the strict access control on PCRs. That is, we should incorporate measurement evidence from lower layers before generating the evidence for higher layers. This suggests a tiered and nested strategy for bundling the evidence. In the case of \mathcal{MS}_1 , to demonstrate the order specified in S_1 , our strategy might produce a collection of quotes of the following form.

$$\begin{aligned} Q_1 &= \llbracket n, p_r, \text{seq}(v_1, v_2) \rrbracket_{sk(t)} \\ Q_2 &= \llbracket n, (p_1, p_2), (\text{seq}(Q_1, v_3), \text{seq}(Q_1, v_4)) \rrbracket_{sk(t)} \\ Q_3 &= \llbracket n, p_{vc}, \text{seq}(Q_2, v_5) \rrbracket_{sk(t)} \end{aligned}$$

The quote Q_1 provides evidence that `rtm` has measured A_1 and A_2 . This quote is itself extended into the PCRs of A_1 and A_2 before they take their measurements and extend the results. Q_2 thus represents evidence that `rtm` took its measurements before A_1 and A_2 took theirs. Similarly, Q_3 is evidence that `vc` took its measurement after A_1 and A_2 took theirs since Q_2 is extended into p_{vc} before the measurement evidence.

Unfortunately, this quote structure is not quite enough to ensure that the proper order is respected. Figure 7 illustrates the problem. In that execution, all the measurements are generated concurrently at the beginning, and each component waits to extend the result until it gets the quote from the layer below. The quotes give accurate evidence for the order in which evidence was *recorded* but not for the order in which the evidence was generated. It must be the job of regular components to ensure that the order of extend events accurately reflects the order of measurement events. We make precise our assumptions for regular components in Section 6. Under those extra assumptions we can prove that a quote generated according to this final strategy is sufficient to ensure that the execution it came from meets the guarantees of Theorem 1.

6 Attestation Systems and Bundling Evidence

In this section we cover the definitions and main result pertaining to the use of TPMS and their use in layered attestations. In order to better focus on the result itself, we omit lemmas and proofs that support it. A brief summary of the relevant definitions is sufficient for the current presentation. The detailed definitions, lemmas, and proofs can be found in the full paper [13].

6.1 Extended Definitions

We first define an *attestation system* to be an augmentation of a measurement system that additionally specifies a set of PCRs P and an assignment L of components to PCRs designating the PCRs into which each component is able to extend. We similarly augment the set of events to include *extend* events $\text{ext}(o, v, p)$ in which component o extends value v into PCR p , and *quote* events $\text{qt}(v, p_I)$ in

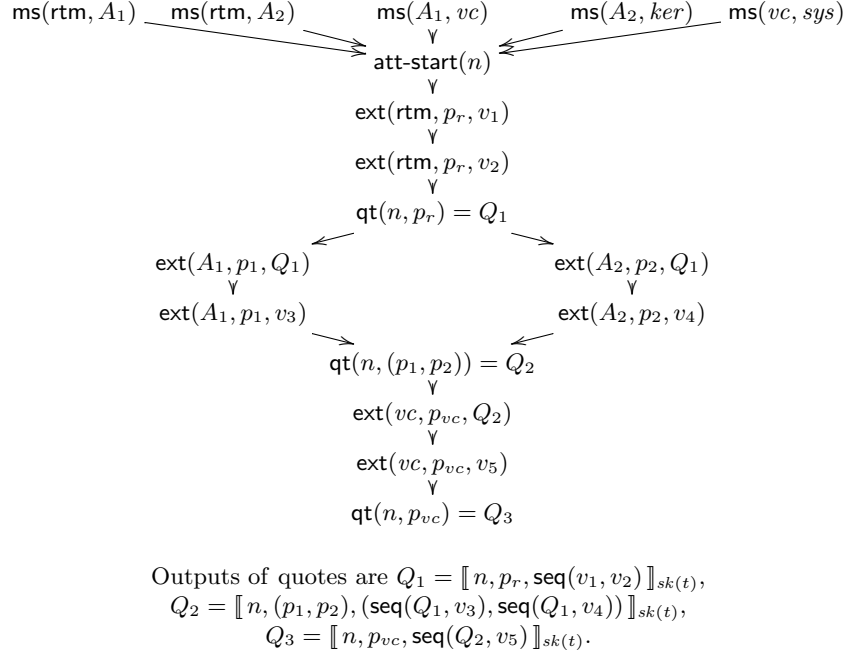


Fig. 7. Defeating Strategy 3

which a TPM produces a signature over some input value v and the contents of PCRs p_I for some index set I .

In order to understand the causal effects of extending values and producing quotes, we must be able to track the contents of any PCR throughout an execution. To that end, we introduce the notion of *extend-ordered* partially ordered sets of events which allows us to unambiguously define the contents of PCRs at relevant events, and require that executions also be extend-ordered. Thus, in executions, we can define the output of a quote event $\text{qt}(v, p_I)$ to be the digital signature $\llbracket v, (p_i)_{i \in I}, (v_i)_{i \in I} \rrbracket_{sk(t)}$ where v_i is the contents of p_i at the quote event, and $sk(t)$ is the uncompromised signature key of the TPM identified by t . For any execution E that has a quote event with output Q , we say that E produces quote Q and write $E \in \mathcal{E}(Q)$.

6.2 Formalizing and Justifying a Bundling Strategy.

We now present an overview of our main results about bundling evidence. We provide the formal statements and discuss their significance. The proofs of these results as well as supporting lemmas can be found in [13].

We begin by formalizing the layered, nested quote structure of Strategy 3 from Section 5. We define the strategy indirectly by first defining a method for

extracting a measurement specification from a quote and then checking whether that specification measures bottom-up.

Bundling Strategy. Let \mathcal{Q} be a set of quotes. We describe how to create a measurement specification $S(\mathcal{Q})$. For each $Q \in \mathcal{Q}$, and each p that Q reports on, and each $v \in \mathcal{MV}(o_2)$ contained in p , $S(\mathcal{Q})$ contains an event $e_v = \text{ms}(o_1, o_2)$ where $M(o_1, o_2)$ and $L(o_1, p)$. Similarly, for each n in the nonce field of some $Q \in \mathcal{Q}$, $S(\mathcal{Q})$ contains the event $\text{att-start}(n)$. Let S_Q denote the set of events derived in this way from $Q \in \mathcal{Q}$. Then $e \prec_{S(\mathcal{Q})} e_v$ iff Q is contained before v and $e \in S_Q$. \mathcal{Q} complies with the bundling strategy iff $S(\mathcal{Q})$ measures bottom-up.

As we saw earlier, this strategy alone is not enough to guarantee to an appraiser that the expected order of measurement actually occurred. This is due to the fact that a quote can only provide direct evidence for the order in which measurements were recorded, not the order in which they were taken. Thus we make two assumptions about executions of attestations systems that will allow us to reconstruct the order of measurement events from the order of corresponding extend events.

Assumption 2 If E contains an event $e = \text{ext}(o, v, p)$ with $v \in \mathcal{MV}(t)$, where o is regular at that event, then there is an event $e' = \text{ms}(o, t)$ such that $e' \prec_E e$. Furthermore, the most recent such event e' satisfies $\text{out}(e') = v$.

This assumption says that when extending measurement values regular components only extend the value they most recently generated through measurement. This might be satisfied in several ways. For example, a system might ensure that measurement results are immediately extended into a PCR, so there is no opportunity for the component or the result to be corrupted between measurement event and extend event. Alternatively, a component may take periodic measurements of its targets only extending the results into a PCR when an attestation is requested. Such an architecture provides more flexibility but would require a mechanism to ensure that the component can reliably retrieve the most recent result of measurement, and that it cannot be altered in the meantime. This suggests that security-focused operating systems, such as SELinux, which enable fine-grained control over the flow of information should play an important part of the architecture of an attestation system.

Assumption 3 Suppose E has events $e \prec_E e'$ where $e = \text{ms}(o_2, o_1)$ and $e' = \text{ext}(o, v, p)$ where $v \in \mathcal{MV}(o_t)$, $o_1 \in D^1(o_t)$. Then either

1. o is corrupt at e' , or
2. there is some $e'' = \text{ms}(o, o_t)$ with $e \prec_E e'' \prec_E e'$.

This assumption is more complex. It says, roughly, for any target o_t , whenever any of its measurers or their contexts are remeasured, then o_t should also be remeasured. This ensures that the most recent measurements of higher layers are at least as recent as the measurements of lower layers. Again, this might be

achieved in a variety of ways, but a natural way would be to use an architecture in which one can specify and enforce fine-grained security policies that can express this constraint.

Assumption 3 can only guarantee that an object is remeasured whenever one of its dependencies is remeasured. It cannot ensure that all orderings of $S(\mathcal{Q})$ are preserved in $\mathcal{E}(\mathcal{Q})$. For this reason we introduce the notion of the core of a bottom-up specification. The *core* of a bottom-up specification S is the result of removing any orderings between measurement events $e_i \prec_S e_j$ whenever e_i is not in the support of e_j . That is, the core of S ignores all orderings that do not contribute to S measuring bottom-up.

We are now ready to show that these assumptions are sufficient to ensure Strategy 3 for bundling evidence is a good one for TPM-based attestations. The following theorem says that if an adversary wants to convince an appraiser that measurements were taken in a given order when in fact they were not, he must perform either a recent or deep corruption.

Theorem 2. *Let $E \in \mathcal{E}(\mathcal{Q})$ such that $S(\mathcal{Q})$ measures bottom-up, and let S' be its core. Suppose that \mathcal{Q} detects no corruptions, and that E satisfies Assumptions 2 and 3. Then one of the following holds:*

1. $E \in \mathcal{E}(S')$,
2. *there is some $o_t \in O$ such that*
 - a. *some $o_2 \in D^2(o_t)$ is corrupted, or*
 - b. *some $o_1 \in D^1(o_t)$ is corrupted after being measured.*

Proof. The proof is provided in [13].

Thus, the adversary cannot escape the consequences of Theorem 1, because anything he does to avoid the conditions of its hypothesis forces him to be subject to its conclusion anyway!

7 Conclusion

In this paper we have developed a formalism for reasoning about layered attestations. Within this framework we have justified the intuition (pervasive in the literature on measurement and attestation) that it is important to measure a layered system from the bottom up (Theorem 1). We also proposed a strategy for using TPMs to bundle evidence. If used in conjunction with bottom-up measurement, we can guarantee an appraiser that if an adversary has corrupted a component and managed to avoid detection, then it must have performed a recent or deep corruption (Theorem 2).

Although we used our model to justify the proposed general and reusable strategies for layered attestations, we believe our model has a wider applicability. It admits a natural graphical interpretation that is straightforward to understand and interpret. Future work to develop reasoning methods within the model could lead to more automated analysis of attestation systems. We believe a tool that

leverages automated reasoning and the graphical interpretation would be a useful asset.

For the present work we made several simplifying assumptions. For instance, we assumed that if measurers (or their supporting components) are corrupted, then they can always forge the results of measurement. This conservative, worst-case view does not account for a situation in which, say, even if the OS kernel is corrupted, it may still be hard to forge the results of a virus scan. Conversely, we also assumed that uncorrupted measurers can always detect corruptions. This is certainly not true in most systems. Adapting the model to account for probabilities of detection would be an interesting line of research that would make the model applicable to a wider class of systems.

Another issue of layered attestations that we did not address here, is the question of what to do when the system components fall into different administrative domains. This would be typical of a cloud architecture in which the lower layers are administered by the cloud provider, but the customers may provide their own set of measurement capabilities as well. A remote appraiser must be able to negotiate an attestation according several policies. Our model might be extended to account for the complexities that arise.

Finally, we chose to study the use of TPMs for bundling evidence. We believe other approaches leveraging timing-based techniques or other emerging technologies including hardware-supported trusted execution environments such as Intel’s new SGX instruction set could be captured similarly. This would allow us to formally demonstrate the security advantages of one approach over another, or understand how to build attestation systems that leverage several technologies.

Acknowledgments

I would like to thank Pete Loscocco for suggesting and guiding the direction of this research. Many thanks also to Perry Alexander and Joshua Guttman for their valuable feedback on earlier versions of this work. Finally, thanks also to Sarah Helble and Aaron Pendergrass for lively discussions about measurement and attestation systems.

References

1. Serdar Cabuk, Liqun Chen, David Plaquin, and Mark Ryan. Trusted integrity measurement and reporting for virtualized platforms. In *Trusted Systems, First International Conference, INTRUST 2009, Beijing, China, December 17-19, 2009. Revised Selected Papers*, pages 180–196, 2009.
2. George Coker, Joshua D. Guttman, Peter Loscocco, Amy L. Herzog, Jonathan K. Millen, Brian O’Hanlon, John D. Ramsdell, Ariel Segall, Justin Sheehy, and Brian T. Sniffen. Principles of remote attestation. *Int. J. Inf. Sec.*, 10(2):63–81, 2011.
3. Intel Corporation. Open attestation. Accessed: 2015-12-16.

4. Anupam Datta, Jason Franklin, Deepak Garg, and Dilsun Kirli Kaynar. A logic of secure systems and its application to trusted computing. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 221–236, 2009.
5. Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 4th ACM Workshop on Scalable Trusted Computing, STC 2009, Chicago, Illinois, USA, November 13, 2009*, pages 49–54, 2009.
6. Stéphanie Delaune, Steve Kremer, Mark Dermot Ryan, and Graham Steel. Formal analysis of protocols based on TPM state registers. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 66–80, 2011.
7. Charles Fisher, Dave Bukovick, Rene Bourquin, and Robert Dobry. SAMSON - Secure Authentication Modules. Accessed: 2015-12-16.
8. Trusted Computing Group. TCG Trusted Network Connect Architecture for Interoperability version 1.5, 2012.
9. Chongkyung Kil, Emre Can Sezer, Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009*, pages 115–124, 2009.
10. Peter Loscocco, Perry W. Wilson, J. Aaron Pendergrass, and C. Durward Mc-Donell. Linux kernel integrity measurement using contextual inspection. In *Proceedings of the 2nd ACM Workshop on Scalable Trusted Computing, STC 2007, Alexandria, VA, USA, November 2, 2007*, pages 21–29, 2007.
11. Richard Maliszewski, Ning Sun, Shane Wang, Jimmy Wei, and Ren Qiaowei. Trusted boot (tboot). Accessed: 2015-12-16.
12. John D. Ramsdell, Daniel J. Dougherty, Joshua D. Guttman, and Paul D. Rowe. A hybrid analysis for security protocols with state. In *Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings*, pages 272–287, 2014.
13. Paul D. Rowe. Principles of layered attestation. *CoRR*, abs/1603.01244v1, 2016. <http://arxiv.org/abs/1603.01244v1>.
14. Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 223–238, 2004.
15. Jinpeng Wei, Calton Pu, Carlos V. Rozas, Anand Rajan, and Feng Zhu. Modeling the runtime integrity of cloud servers: A scoped invariant perspective. In *Cloud Computing, Second International Conference, CloudCom 2010, November 30 - December 3, 2010, Indianapolis, Indiana, USA, Proceedings*, pages 651–658, 2010.

Measuring Protocol Strength with Security Goals

Paul D. Rowe · Joshua D. Guttman · Moses D. Liskov

the date of receipt and acceptance should be inserted later

Abstract Flaws in published standards for security protocols are found regularly, often after systems implementing those standards have been deployed. Because of deployment constraints and disagreements among stakeholders, different fixes may be proposed and debated. In this process, security improvements must be balanced with issues of functionality and compatibility.

This paper provides a family of rigorous metrics for protocol security improvements. These metrics are sets of first order formulas in a goal language $\mathcal{GL}(\Pi)$ associated with a protocol Π . The semantics of $\mathcal{GL}(\Pi)$ is compatible with many ways to analyze protocols, and some metrics in this family are supported by many protocol analysis tools. Other metrics are supported by our Cryptographic Protocol Shapes Analyzer CPSA.

This family of metrics refines several “hierarchies” of security goals in the literature. Our metrics are applicable even when, to mitigate a flaw, participants must enforce policies that constrain protocol execution. We recommend that protocols submitted to standards groups characterize their goals using formulas in $\mathcal{GL}(\Pi)$, and that discussions comparing alternative protocol refinements measure their security in these terms.

1 Introduction

Security standards often contain flaws, and therefore evolve over time as people correct them. Often, these flaws are discovered after deployment, which puts pressure on the choice of mitigation. The constraints of operational deployments and the needs of the various stakeholders are crucial, but so is understanding the significance of the attack.

How are we to choose among the alternatives proposed to repair a flaw? A security flaw is a failure of the protocol to meet a goal—often one not well understood until after the flaw becomes apparent—and a revised understanding of the goals of the protocol is necessary to ensure that the mitigation is secure. Obviously, satisfying a goal that was not previously met is essential to any mitigation. However, two alternatives may eliminate the same insecure scenario while differing in the security they provide.

In this paper, we describe a formal language for expressing protocol security goals. We use sets of formulas in this language to compare protocols. A protocol Π_2 is then at least as secure as another protocol Π_1 with respect to a set of goal formulas G if, for each formula $\Gamma \in G$, if Π_1 achieves the goal Γ , then so does Π_2 . We will explore various sets of formulas G below. We will argue that any set G containing formulas of a reasonable syntactic form should be regarded as a “measurement” for the security of the protocols.

In the case of a singleton $G = \{\Gamma\}$, we can use a variety of protocol analysis tools such as Maude-NPA [18], ProVerif [8], Scyther [14], and our Cryptographic Protocol Shapes Analyzer CPSA [32] to ascertain whether Π_2 is at least as secure as Π_1 with respect to G . It suffices to show that Π_2 achieves the goal Γ , or else that some attack on Π_1 provides a counterexample showing that Π_1 does not achieve Γ . In the same spirit, we can use the same tools to measure protocols relative to two-element sets $G = \{\Gamma_1, \Gamma_2\}$ or other finite sets. We simply evaluate the protocols for each goal separately and combine the results.

CPSA also allows protocol comparisons using larger sets G , especially sets of implications $\Phi \implies \Psi$ that all share the same hypothesis Φ .

Measuring security. To compare alternative fixes, we need to measure the security of the alternatives. We view their security as their power to exclude failures. This suggests the basic tenet:

Tenet: A system S_2 is *at least as secure as* S_1 , written $S_1 \trianglelefteq S_2$, if and only if any attack that is successful against S_2 is also successful against S_1 .

Of course, it is difficult to instantiate this tenet in general. It requires a robust and realistic model of the adversary’s capabilities, and it requires a clear understanding of what constitutes an attack. We will think of attacks as possible goals that fail, rather than as meaning the specific bad executions that illustrate how they can fail.

Indeed, S_1 and S_2 have to be sufficiently similar to make sense of the “same” attack succeeding on each system. Nevertheless, we believe this tenet can be used to clarify why some security metrics provide more insight than others. Let M be some measurement technique that yields values in a domain ordered by \leq . An ideal property to strive for in measuring security is the following.

$$M(S_1) \leq M(S_2) \text{ iff } S_1 \trianglelefteq S_2 \quad (1)$$

The crux here is a general philosophy of measurement, according to which the outcome of measurement under the \leq ordering should be a perfect substitute for the \trianglelefteq ordering applied directly to the systems. Luce and Suppes [27] refer to this as the Axiom of Order for theories of measurement. In practice, we may have to weaken property (1) in several ways. For instance, we might only achieve the property if we relativize \trianglelefteq to a restricted class A of attacks and adversaries yielding a restricted security ordering \trianglelefteq^A . But it serves to explain why certain metrics do not measure security well. Let’s consider a simple example.

The number of lines of code tends to be correlated with the number of bugs in software, and hence to the number of security vulnerabilities. If we let M be a method that counted the number of lines of code and output the negative of that number, then $M(S_1) \leq M(S_2)$ would mean that S_1 has at least as many lines of code as S_2 . But it certainly does not follow that S_2 is more secure than S_1 . In this instance $M(S_1) \leq M(S_2)$ is only loosely correlated with $S_1 \trianglelefteq S_2$, and so the metric is not very reliable.

The lines-of-code metric also has a property that is common to many security metrics proposed in the literature: it takes values in the real numbers. The urge to have *quantitative* metrics is understandable, especially when striving to achieve the rigor of measurement in the physical sciences. Analogies to length, weight, temperature, etc. make quantitative metrics almost irre-

sistible. But our basic tenet shows why this approach cannot work in general. If the \trianglelefteq security ordering naturally corresponds to sets of attacks ordered by inclusion, only very rarely will systems be totally ordered by \trianglelefteq . In many cases S_1 will admit attacks that S_2 does not allow and vice versa, meaning \trianglelefteq tends to be only a partial order. Any metric that tries to totally order systems is very likely not to achieve property (1).

This paper represents an instantiation of our general philosophy of measuring security. We recognize that each time there is an attack on a protocol that means there is some (possibly unstated) security goal the protocol does not achieve. Definition 3 in Section 3.3 is a reformulation of our basic tenet.

By *security goal*, we will mean logical formula of a particular form. Roughly speaking, a security goal is an implication $\Phi \implies \Psi$ where Φ is a conjunction of atomic formulas, and Ψ is built from atomic formulas by conjunction, disjunction, and existential quantification. (See Def. 1 in Section 3.1). When G is a set of formulas of this form, we will write $\Pi_1 \trianglelefteq^G \Pi_2$ to mean that, for every formula $\Gamma \in G$, if Π_1 achieves Γ , then so does Π_2 . When G is a singleton, several protocol analysis tools can assist in determining whether $\Pi_1 \trianglelefteq^G \Pi_2$.

Enrich-by-need. Enrich-by-need analysis—as in the Cryptographic Protocol Shapes Analyzer (CPSA) [32] or in Scyther [14]—allows us to resolve \trianglelefteq^G for some important non-singleton sets G .

Each enrich-by-need analysis process starts from a *scenario* containing some protocol behavior, such as one or more participant’s local runs, and also some assumptions about freshness and uncompromised keys. The analysis returns a set of result scenarios that show all of the minimal, essentially different ways that the starting scenario could happen. When the starting scenario is undesirable (e.g. a confidentiality failure), we would like this to be the empty set. When the starting scenario is the behavior of one principal, then the analysis finds what the protocol guarantees from that participant’s “point of view.” These results indicate what authentication guarantees the protocol provides to that party.

For each run of the analyzer, there is a security goal formalizing the result of the analysis. Its hypothesis Φ describes the content of the starting scenario, and its conclusion is a disjunction. Each disjunct describes the content of one minimal, essentially different execution containing the starting point. Thus, this is a strongest security goal with hypothesis Φ .

Specifically, this goal is strongest in the *implication* ordering, where $\Gamma_2 \implies \Gamma_1$ means that Γ_1 is below Γ_2 in this ordering. Enrich-by-need analysis computes maximal elements for certain subclasses G of security goals. Enrich-by-need can act as our measurement technique,

M , satisfying property (1) relative to the security ordering \trianglelefteq^G . In Section 6.3 we will identify key subclasses G of goals, defined in equation (9), for which each protocol achieves a unique strongest goal (Theorem 2). The maximal elements identified by the enrich-by-need method allow us to measure and compare protocol security relative to \trianglelefteq^G (Corollary 1).

Application to standards. A concrete formal language of security goals makes security claims explicit and verifiable. One reason flaws and weaknesses are repeatedly discovered in published standards is that the standards include vague statements, for example about two parties achieving “authentication,” without explaining what that means in practice. Basin et al. [5] have illustrated the problems that can arise when standards do not contain any claims about what properties they achieve or under which adversary models. It is thus important for the standardization process to create concrete artifacts that serve as evidence of the security of a protocol design.

Our logical language of security goals supports the goals of [5]. We envision a process where those proposing a protocol are required to include explicit and formal claims about what security properties it achieves. These claims should be explicit enough to allow formal, independent verification that the protocol achieves those security properties. The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) have started to lay the technical groundwork for such a process by publishing a standard (ISO/IEC 29128 [19]) for certifying the design of cryptographic protocols. While this is a good start, it still leaves room for ambiguity based on the specific formalism chosen. Our language is designed to be independent of both the underlying formalism used to represent the protocol and the tool(s) used to verify the claim. This allows a proposal to be verified relative to the same claim by independent experts using different tools.

Despite such efforts to avoid protocol weaknesses early in the standardization process, flaws will still undoubtedly be discovered after publication and deployment. Our development here of a hierarchy of security goals \trianglelefteq^G against which to measure security is useful for this purpose. By expressing the flaw as some (previously unstated) goal that is not achieved, standards committees can more easily identify and compare related goals. Having an objective measure of the relative strength of security goals will allow the committee members to separate the security properties from other factors that may affect the feasibility of certain mitigations. The use of enrich-by-need analysis is particularly appropriate in this case, because the analysis can com-

pare the strength of two proposed mitigations relative to an *infinite* set of goals that share some hypothesis Φ . Since the analysis results in the strongest security goal with Φ as a hypothesis, it does not require as much ingenuity on the part of the committee members to identify a particular strengthened security goal. This is an advantage, especially since flaws are frequently already the result of the failure of imagination in the human designers to identify useful security goals.

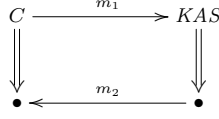
Our contributions. We make three main contributions.

1. We introduce the metrics of security \trianglelefteq^G for cryptographic protocols. Each metric is parameterized by some set of desired security goals G .
2. We identify the sets G for which the enrich-by-need analysis method can act as a measurement tool M to compare the security of two or more protocols relative to \trianglelefteq^G .
3. We ground our theory in protocols that have undergone standardization. We compare our sets G with security goals discussed in the literature. Our methods can guide standards bodies’ deliberations on proposed designs and protocol improvements.

Structure of this paper. In order to help the reader gain intuition for when and why it is useful to measure and compare the security of protocols, we begin in Section 2 by considering the Kerberos public key extension PKINIT. The initial version contained a flaw allowing a man-in-the-middle attack. Two alternatives emerged to fix this. We build up some intuition about how to formally express the security goals not met by the flawed version. Section 3 develops our logical language of security goals and explains how sets of goals can serve as security metrics.

Having explained our central ideas, we discuss related work in Section 4. We explain, in depth, the connection between our logical security goals related by implication and well-studied authentication hierarchies. We show how our formalism encompasses and extends that previous work. In Section 5, we show that our logical language helps us to understand how policy considerations that sit outside a protocol proper can contribute to the protocol’s security. In Section 6 we explain the enrich-by-need method with examples, and prove that it allows us to compare protocols with regard to some infinite sets of goals. Concluding thoughts are in Section 7.

This paper is an extended version of our [23].



$$m_1 = [t_C, n_2]_{\text{sk}(C)}, C, T, n_1$$

$$m_2 = \{\{[k, n_2]_{\text{sk}(S)}\}_{\text{pk}(C)}, C, TGT, \{AK, n_1, t_S, T\}_k\}_k$$

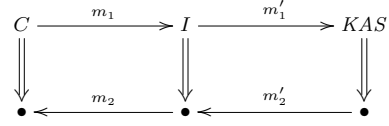
Fig. 1 PKINIT version 25, where $TGT = \{AK, C, t_S\}_{k_T}$

2 Example: Kerberos PKINIT

PKINIT [39] is an extension to Kerberos [30] that allows a client to authenticate to the Kerberos authentication server (KAS) and obtain a ticket-granting ticket using public-key cryptography. This is intended to avoid the management burden of establishing and maintaining user passwords, which the standard Kerberos exchange requires.

Cervesato et al. [11] found a flaw in PKINIT version 25, which was already widely deployed. The flaw was eventually fixed in version 27. Figure 1 shows the expected message flow between the client and the KAS in v. 25. The client provides a KAS with its identity C , the identity T of the server it would like to access, and a nonce n_1 . It also includes a signature over a timestamp t_C and a second nonce n_2 using the client's private key $\text{sk}(C)$. The KAS with identity S replies by creating a fresh session key k , signing it using its key $\text{sk}(S)$ together with the nonce n_2 and encrypting the signature using the client's public key $\text{pk}(C)$. It uses the session key k to protect another session key AK to be used between the client and the subsequent server T , together with the nonce n_1 and an expiration time t_S for the ticket. The ticket TGT is an opaque blob from the client's perspective because it is an encryption using a key shared between S and T . It contains AK , the client's identity C and the expiration time t_S of the ticket.

The flaw. In Cervesato et al.'s attack [11] (Fig. 2), an adversary I has obtained a private key to talk with the KAS S . I uses it to forward any client C 's initial request, passing it off as a request from I . I simply replaces C 's identity with I 's own, re-signing the timestamp and nonce n_2 . When the S responds, I re-encrypts the response for C , this time replacing the identity I with C . In the process, the adversary learns the session key k , and thus can also learn the subsequent session key AK . This allows the attacker to read any subsequent communication between the client and the next server T . Moreover, the adversary may impersonate the ticket granting server T to C , and *vice versa*, because



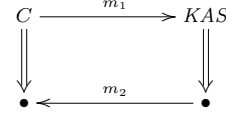
$$m_1 = [t_C, n_2]_{\text{sk}(C)}, C, T, n_1$$

$$m_1' = [t_C, n_2]_{\text{sk}(I)}, I, T, n_1$$

$$m_2 = \{\{[k, n_2]_{\text{sk}(S)}\}_{\text{pk}(C)}, C, TGT, \{AK, n_1, t_S, T\}_k\}_k$$

$$m_2' = \{\{[k, n_2]_{\text{sk}(S)}\}_{\text{pk}(I)}, I, TGT, \{AK, n_1, t_S, T\}_k\}_k$$

Fig. 2 Attack on flawed PKINIT, where $TGT = \{AK, I, t_S\}_{k_T}$



$$m_1 = [t_C, n_2]_{\text{sk}(C)}, C, T, n_1$$

$$m_2 = \{\{[k, \mathbf{F}(C, n_2)]_{\text{sk}(S)}\}_{\text{pk}(C)}, C, TGT, \{AK, n_1, t_S, T\}_k\}_k$$

Fig. 3 Generic fix for PKINIT

the participants rely on the protocol to ensure that they are the only entities with knowledge of AK .

The attack arises from a lack of cryptographic binding between the session key k , and the client's identity C [11]. After C 's two-message exchange, she knows the KAS produced the keying material k recently, because of its binding with n_2 . However, the KAS S may not have intended k for C .

The protocol revision process. Since this absence of C 's identity is the root cause of the attack, the natural fix is to include C as an additional field in the signed portion of the second message. Indeed this is the first suggestion in [11].

The authors of the PKINIT standard offered a different suggestion. For operational feasibility—namely, to preserve the previous message format—more than security, the PKINIT authors suggested replacing n_2 with a message authentication code over the entirety of the first message, keying the MAC with k . Since the client's identity is contained in the first message, this proposal also creates the necessary cryptographic binding between k and C , as well as with n_2 .

Cervesato et al. used a manual proof method to verify a generic scheme for mitigating the attack (Fig. 3), ensuring that the two proposals were instances of the scheme. This allowed them to avoid the time-consuming process of writing proofs for any other proposals that might also fit this scheme. They verified that the attack fails if n_2 is replaced with any expression $F(C, n_2, \dots)$ that is *injective* on C and n_2 ; that is, $F(C, n_2, \dots) = F(C', n_2', \dots)$ implies $C = C'$ and $n_2 = n_2'$.

We obtain the first proposal by instantiating F as concatenation: $F(C, n_2) = C, n_2$. The second proposal instantiates F as the MAC of the client’s request:

$$F(C, n_2, \dots) = H_k([t_C, n_2]_{\text{sk}(C)}, C, T, n_1).$$

Since the MAC provides second preimage resistance, the injectivity requirement will hold, with overwhelming probability, in any execution the adversary can engineer.

Security goals. The PKINIT parable illustrates recurring themes in developing and maintaining protocol standards. An attack often shows us that we care about previously unstated and unrecognized security goals as observed by Basin et al. [5]. PKINIT achieves some level of authentication, but it fails a more stringent type of authentication. In Lowe’s terms [26], it achieves *recent aliveness* both for the client and for the KAS S , because each party signs time-dependent data. However, PKINIT does not achieve weak agreement, since C does not know that S was engaged in the protocol *with* C . The attack helps us to express the goal that the flawed protocol does not meet.

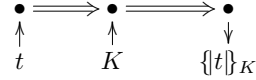
But an attack itself does not uniquely identify a security goal. We learned that it is important for the client to be guaranteed that it agrees with S on the client’s identity, but what about other values such as the expiration time of the ticket? Operational difficulties might arise if the client is unaware of this expiration time, but are there any security consequences? Indeed a key contribution of [11] is to state carefully what security goal the repair provides.

This goal can be achieved in different ways. Different stakeholders may prefer different mitigations because of issues of efficiency, ease of deployment, or robustness to future modification. In PKINIT, the researchers opted for a change that was minimally invasive to their formal representation, thereby highlighting the root cause of the problem. The protocol designers had more operational context to constrain the types of solutions they deemed feasible.

While a pair of choices might both manage to satisfy some stated security goal, one of them may actually satisfy strictly stronger goals than another. We propose a goal language (Section 3) to express when a protocol mitigation is at least as good as a competitor—or strictly better than it—from the security point of view.

Strand spaces. We will develop our ideas using the strand space terminology [37, 20]. A *strand* is a sequence of transmission and reception events, each of which we will call a *node*. We use strands to represent the behavior of a single principal in a single local protocol session. We call these *regular strands*. We also use strands

to represent the basic abilities of the adversary. For instance, a strand representing the adversary’s ability to encrypt contains two reception nodes, in which the plaintext t and the encryption key K are received, followed by a transmission node in which the encryption is sent:



By convention, we draw strands with double-arrows connecting the successive nodes $\bullet \Rightarrow \bullet$, and single arrows indicating the message flow $\bullet \rightarrow \bullet$.

In this framework an execution is a kind of directed graph with these two kinds of edges. These graphs are *bundles*, meaning all finite directed acyclic graphs where (i) the nodes at the two ends of a message transmission arrow are labeled with the same message; (ii) each reception node has exactly one incoming message; and (iii) when a node on a strand is included in the graph, then so are all its predecessors on that strand. However, a bundle does not have to run all of the strands “to completion,” and it may therefore contain only an initial segment of the nodes of that strand.

A *protocol* Π is a finite set of strands, called the *roles* of Π , together with possibly some auxiliary assumptions about fresh and non-compromised values. The messages sent and received on these strands contain *parameters* such as $C, S, AK, n_1, n_2, \dots$ in PKINIT. The *regular strands* of Π consist of all strands obtained from the roles of Π by applying substitutions to these parameters. A Π -bundle is a bundle where every strand is either an adversary strand or a regular strand of the protocol Π .

Fig. 2 becomes a PKINIT-bundle when we expand the single central strand labeled I into a collection of adversary strands. We consider Fig. 2 an informal shorthand for the resulting bundle.

Formalizing the authentication goal. The attack of [11] undermines what the client C should know when he has completed a local run of PKINIT. The client knows less about what the KAS S has done than expected.

The actions of the regular (non-adversary) principals are *message transmission nodes* and *message reception nodes*. We will formulate the client’s expectation about the KAS’s behavior as a formula about the transmission and reception nodes of the principals.

The formula applies in a situation, depicted in (2), where there is a reception node n which completes a run

of the client role, which we will write $\text{ClientDone}(n)$.

$$\begin{array}{c} \text{sk}(S) \in \text{non} \quad \bullet \longrightarrow \\ s_1 \Downarrow \\ n \longleftarrow \\ s_1 \in \text{Client}[C, S, _, _, _, _, _, _, _] \end{array} \quad (2)$$

This node n belongs to a run of the protocol in which the active principal has some identity C , and its intended peer is a Kerberos Authentication Server S . We will write this $\text{Self}(n, C) \wedge \text{Peer}(n, S)$. Of course, if S 's signature key $\text{sk}(S)$ is compromised, then C learns nothing from a run of the protocol. Thus, we will assume that it is uncompromised, written as $\text{Non}(\text{sk}(S))$. For the moment, we ignore the other parameters of the Client role, since we will not assume anything about them.

We regard these formulas as forming a hypothesis, when combined by conjunction. Thus, we would like to understand what must be true when this hypothesis holds:

$$\begin{aligned} \text{ClientDone}(n) \wedge \text{Self}(n, C) \\ \wedge \text{Peer}(n, S) \wedge \text{Non}(\text{sk}(S)) \end{aligned} \quad (3)$$

In the attack, there is a local run of the KAS role, and in fact the server has the intended identity S . The problem is that the server's intended peer is not the client C , but some compromised client I . Thus, the behavior in Fig. 2 is a counterexample to the goal:

$$\begin{aligned} \forall n, C, S. (3) \implies \exists m. \text{KASDone}(m) \\ \wedge \text{Self}(m, S) \wedge \text{Peer}(m, C) \end{aligned} \quad (4)$$

where the whole of formula (3) is the hypothesis, although we have contracted it to save space. Here also we write $\text{KASDone}(m)$ to indicate that the transmission node m is the final node of the KAS 's role. We again use Self to refer to the identity of the participant enacting this role, and Peer to refer to its intended partner.

Clearly this is a weak goal, since it says nothing about other parameters such as the nonces n_1, n_2 , the session key k , or the server T that the ticket will be shared with. But Fig. 2 is also a counterexample to all its stronger goals.

A curious fact about this formalism is that it says nothing about the specific messages sent. It talks about the nodes, such as n, m , and asserts that they lie on a particular role at a specific position in the sequence of nodes in which the role engages. It talks about the parameters associated with the nodes. For instance, the Peer parameter of n is the same identity S whose signature key is assumed to be uncompromised. Moreover, this is the same as the Self parameter of some node m , and that m is a KASDone node.

However, it never assumes or asserts that the messages formed from the parameters have a particular layout or structure. This means that the same formula can describe executions in different protocols.

For instance, it is clear how to interpret formula (4) in the two revised versions of PKINIT. There is a self-explanatory convention that links protocols in PKINIT v. 25 to roles in its two candidate successors, and similarly for their parameters. When the protocols are more remotely related, Guttman's formal notion of translation applies [22]. Our convention allows us to use the identity translation a large range of cases.

Formalizing a non-disclosure goal. We can represent secrecy goals in a similar style, using a special auxiliary role which we can assume belongs to all protocols implicitly. This is the "listener role" that consists of a single reception node. It has a single parameter x , and the message received on this node is x . It represents the assumption that x is compromised, and observed unprotected. We write $\text{Lsn}(n)$ to express that n is the reception node lying on an instance of the listener role. We write $\text{Hear}(n, x)$ to express that the message heard on node n is x , i.e. to stipulate a value for the parameter of the role. Thus, consider the assumption that adds a listener to formula (3):

$$\begin{aligned} \text{ClientDone}(n) \wedge \text{Self}(n, C) \wedge \text{SessKey}(n, k) \\ \wedge \text{Peer}(n, S) \wedge \text{Non}(\text{sk}(S)) \\ \wedge \text{Lsn}(m) \wedge \text{Hear}(m, k) \end{aligned} \quad (5)$$

Here we are also using the predicate $\text{SessKey}(n, k)$ to refer to the session key parameter of the client's final reception node. The formula asserts that the same value k is also heard unprotected on the listener node m . This formula represents a situation visualized in (6).

$$\begin{array}{c} \text{sk}(S) \in \text{non} \quad C \longrightarrow \\ s_1 \Downarrow \quad n \longleftarrow \quad \xrightarrow{k} m \\ s_1 \in \text{Client}[C, S, _, _, _, _, _, k, _] \end{array} \quad (6)$$

For the protocol to ensure secrecy for k in this situation means that this situation should never be able to arise. Here we interpret secrecy failures as meaning the full disclosure of a secret. This is coarser than the standard cryptographic definition, which refers to any ability of the adversary to distinguish the secret from a random value [6]. Formalizing secrecy as full disclosure, the security goal would be:

$$\forall n, m, C, S, k. (5) \implies \text{false} \quad (7)$$

Unfortunately, the adversary can extract k from m'_2 in the run shown in Fig. 2. Thus, Fig. 2 illustrates why this goal fails: The adversary has the power to transmit k so that it will be heard on a listener node.

Thus, both non-disclosure and authentication goals are expressible using these ideas.

3 Protocol Goals to Measure Security

As we have just illustrated, we express protocol goals as formulas in first order logic. For each protocol Π , there is a goal language $\mathcal{GL}(\Pi)$; however, these languages are designed so that for related protocols, the languages can be similar or often identical. This helps when comparing the goals achieved by related protocols.

3.1 The Goal Languages

The goal language is designed to have the minimum possible expressiveness while remaining useful. It contains no arithmetic; it contains no inductively defined data-types such as terms; and it has no ability to describe the syntax of messages.

This is an important and useful feature of the goal language: goals are much easier to view as logical objects that exist independent of a particular protocol when they do not reference the syntax or structure of messages. The ability to view goals independently from individual protocols is essential when trying to compare different protocols in terms of the goals they achieve. This reduced expressiveness has proved useful in prior work. Formulas in $\mathcal{GL}(\Pi)$ are preserved under a class of “security preserving” transformations between protocols [22]. Also, for an interesting restricted class of protocols, Dougherty and Guttman showed the set of security goals they achieve is decidable [16]. Mödersheim et al. [2] show how to adapt another analysis approach to this class of goals.

Predicates in the goal language. We will describe the goal language $\mathcal{GL}(\Pi)$ associated with a given protocol Π . Although there is a connection between $\mathcal{GL}(\Pi)$ and Π , our intention is to be able to describe logical sentences that apply both to Π and to variants of Π .

For each node in a role of Π , the goal language $\mathcal{GL}(\Pi)$ has an associated *role position predicate*. The two predicates $\text{ClientDone}(n)$ and $\text{KASDone}(m)$ used above are examples. Each role position predicate is a one-place predicate that says what kind of node its argument n, m refers to.

On each node, there are parameters. The *parameter predicates* are two place predicates. Each parameter predicate associates a node with one of the values that has been selected when that node occurs. For instance, $\text{Self}(n, c)$ asserts that the *self* parameter of n is c . This allows us to assert agreement between different strands.

Functions:	$\text{pk}(a)$ $\text{ltk}(a, b)$	$\text{sk}(a)$	$\text{inv}(k)$
Relations:	$\text{Preceq}(m, n)$ $\text{Unq}(v)$	$\text{Coll}(m, n)$ $\text{UnqAt}(n, v)$	$=$ $\text{Non}(v)$

Table 1 Protocol-independent vocabulary of languages $\mathcal{GL}(\Pi)$

$\text{Peer}(m, c)$ asserts that m appears to be partnered with c , which is the same principal who is in fact the *self* parameter of n .

The role position predicates and parameter predicates vary from protocol to protocol, depending on how many nodes the protocol has, and how many parameters. However, when we regard two protocols as variants of each other, we expect a certain amount of overlap between the associated names. For instance, the $\text{ClientDone}(n)$ predicate in the example should have a meaning (i.e. a node that satisfies it) in any variant of the PKINIT protocol. Furthermore, we expect that the satisfying nodes represent informally corresponding activity in the two variants.

Beside role position predicates and parameter predicates, $\mathcal{GL}(\Pi)$ has the protocol-independent vocabulary in Table 1. It helps to express the structural properties of bundles. $\text{Preceq}(m, n)$ asserts that either m and n represent the same node, or else m occurs before n ; $\text{Coll}(m, n)$ says that they lie on the same strand. $m = n$ is satisfied when m and n are equal. $\text{Non}(v)$ and $\text{Unq}(v)$ express non-compromise and freshness (unique origination), and $\text{UnqAt}(n, v)$ identifies the node at which v is assumed fresh. $\text{pk}(a)$ and $\text{sk}(a)$ relate a principal a to its keys, $\text{ltk}(a, b)$ represents the long-term shared key of two principals a, b , and $\text{inv}(k)$ is the inverse of a key.

We have only presented here an informal idea of how to interpret formulas in this language. For a full, formal description of the semantics of the satisfaction relation \models see [22].

The examples of the last section illustrate how we can use the vocabulary of $\mathcal{GL}(\Pi)$ to express a variety of security goals. These include authentication goals and—since we assume that any protocol Π contains the listener role—non-disclosure goals as well.

Goals. The formulas that we used in our examples have a special form. They are implications. Their hypotheses are conjunctions of atomic formulas of $\mathcal{GL}(\Pi)$. The conclusions took two superficially different forms. Formula (4) has an existential formula as its conclusion. It asserts that an additional event exists, satisfying a particular role position predicate, and with some parameters matching those in the hypothesis. Formula (7) has the conclusion **false**. A conclusion could also be a disjunction, where the protocol allows the behavior assumed in the hypothesis to be explained in a number

of different ways. For instance, the *KAS* server may have executed either the role shown in Fig. 1, or else a different role in which it first retrieves a public-key certificate for C , and then replies with the message shown in Fig. 1. Since we may regard **false** as the degenerate disjunction with zero disjuncts, and the conclusion of formula (4) as a degenerate disjunction with one disjunct, we regard them all as having n -ary disjunctions as conclusions. Since the goals are intended to hold in all cases, we regard any variables free in the whole formula as implicitly universally quantified. Thus, we stipulate:

Definition 1 A *security goal* (or sometimes simply a *goal*) is a closed formula $\Gamma \in \mathcal{GL}(\Pi)$ of the form

$$\forall \bar{x}. (\Phi \implies \bigvee_{1 \leq j \leq i} \exists \bar{y}_j. \Psi_j) \quad (8)$$

where Φ and Ψ_j are conjunctions of atomic formulas. We write

$$\begin{aligned} \text{hyp}(\Gamma) &= \Phi, \text{ and} \\ \text{conc}(\Gamma) &= \bigvee_{1 \leq j \leq i} \exists \bar{y}_j. \Psi_j. \end{aligned}$$

We assume that the existentially bound variables in \bar{y}_j are distinct from all variables in $\text{hyp}(\Gamma)$; this is no loss of generality, since we can rename bound variables.

Non-disclosure goals here are the special case in which the disjunction is empty since the upper index is 0.

We propose to express the security services that protocols provide by a set of formulas of the form (8). These so-called *geometric sequents* are natural to express security goals. Each enumerates some finite number of facts that serve as the hypothesis. Then, the claim in the conclusion is that one of zero or more alternatives holds, where each of these is a finite number of facts that should also be found to hold. Thus, the claim is localized, and independent of the totality of behavior in the world of users of the protocol. This is quite appropriate for security properties.

We do not formalize here indistinguishability properties, which are properties of pairs of runs, or probabilistic properties, which focus on the distributions governing runs.

3.2 Measuring a Protocol with Goal Formulas

Security goals admit a natural partial order based on implication: $\Gamma_1 \leq \Gamma_2$ iff $\Gamma_2 \Rightarrow \Gamma_1$. Stronger goals are higher in the partial order. We can use this partial order as a measure of the security of a protocol in the following way.

Definition 2 A protocol Π *achieves* security goal Γ iff for every bundle \mathcal{B} of Π , $\mathcal{B} \models \Gamma$.

This provides our formal basis for deciding whether a protocol is “good enough” for a given purpose. An immediate consequence of this definition is that achieving goals is downward closed in the partial order. More formally:

Lemma 1 *If Π achieves Γ' and $\Gamma \leq \Gamma'$, then Π also achieves Γ .*

Thus, if Π achieves some goal Γ , it also achieves any other goal that is a consequence of Γ .

Def. 2 is intuitively clear. However, it quantifies over all bundles \mathcal{B} . This makes it look like a daunting definition to verify in practice. Although this problem is known to be undecidable in general [17], there is a semi-decision procedure for finding counterexamples.

Theorem 1 *There is a semi-algorithm to find a bundle \mathcal{B} of Π such that $\mathcal{B} \not\models \Gamma$, if any exists.*

Proof Let $\text{conc}(\Gamma)$ be $\bigvee_{1 \leq j \leq i} \exists \bar{y}_j. \Psi_j$. Since Π -bundles are finite structures, we can enumerate them. For each \mathcal{B} , again because it is a finite structure, there are at most finitely many variable assignments η such that $\mathcal{B} \models_\eta \text{hyp}(\Gamma)$.

For each such η and each j such that $1 \leq j \leq i$, there are only finitely many ways to extend η to assign values to the variables \bar{y}_j in \mathcal{B} . Thus, we can determine whether η satisfies $\exists \bar{y}_j. \Psi_j$. \square

Of course, the method detailed in the proof above is naively inefficient. Numerous tools exist that implement much more efficient algorithms to achieve the same result. Throughout this paper we use our tool CPSA to implement this check.

We would like to apply these ideas to PKINIT v. 25 and the two proposed mitigations from Section 2. Let us use PKINIT₁ to denote the fix in which $F(C, n_2) = (C, n_2)$, and use PKINIT₂ to denote the other fix in which $F(C, n_2) = H_k([t_C, n_2]_{\text{sk}(C)}, C, T, n_1)$. In the previous section we identified formula (4) as the security goal that is not achieved in PKINIT v. 25. Call this formula Γ . Cervesato et al. prove by hand that both PKINIT₁ and PKINIT₂ achieve Γ , so it should be no surprise that applying CPSA to those versions confirms the result. Thus both of these fixes are good enough for the newly described goal Γ .

3.3 Comparing Protocols

We want to use security goals not only to measure a protocol against a given goal, but also to measure the relative strength of two or more protocols Π_i against each other. We will be able to do this, but only when

the role position predicates and parameter predicates have a well-defined semantics for each Π_i . This ensures that if Γ is a goal then we can ask whether $\mathcal{B} \models \Gamma$ for any bundle \mathcal{B} of any of the protocols Π_i .

There is a certain amount of freedom in choosing the names of these predicates, and the results of the comparisons will depend on the names chosen. We may arrive at strange conclusions if, for example, in PKINIT_1 , $\text{ClientDone}(\cdot)$ is satisfied by the last node on a client strand, but in PKINIT_2 it is satisfied by the last node of a server strand. There is a natural way to identify the nodes and parameters of one protocol with those of its variants when there are protocol transformations [22] between them. For example, all the variants of PKINIT have corresponding nodes and parameters. The only thing that changes is the exact structure of the messages. [22] formalizes what it means for the semantics of a language to respect these identifications. For the remainder of this section we assume our security goals are expressed in a goal language where the predicate names are uniformly chosen for each of the protocols Π_i , respecting the natural identifications of nodes and parameters.

The partial order \leq on goals naturally induces a partial order \preceq on protocols in the following way.

Definition 3 A protocol Π_2 is *at least as strong as* a protocol Π_1 , written $\Pi_1 \preceq \Pi_2$, iff for every goal Γ , if Π_1 achieves Γ then so does Π_2 .

For the set of all goals that the protocol Π achieves, we write:

$$\text{Ach}(\Pi) = \{\Gamma \mid \Pi \text{ achieves } \Gamma\}.$$

The position of each Π in the partial order \preceq is determined by the set $\text{Ach}(\Pi)$ of goals it achieves. Then $\Pi_1 \preceq \Pi_2$ iff $\text{Ach}(\Pi_1) \subseteq \text{Ach}(\Pi_2)$. So \preceq mirrors the partial order on sets of goals ordered by inclusion. Thus, the \preceq order justifies us in regarding Ach as a measure of protocol strength, as in our Tenet for measurement (cf. Eq. 1). By Lemma 1, the sets $\text{Ach}(\Pi)$ are closed under logical implication.

Definition 3 is a very strong condition because it accounts for *every* goal. Indeed, the condition may be too strong for practical use. For one thing, it is not clear how one would verify that $\Pi_1 \preceq \Pi_2$. That would require a structured way to consider every goal, or at least an efficient way of calculating $\text{Ach}(\Pi)$. Also, protocols are typically designed with a small, finite number of goals in mind. It may be sufficient to understand the relative behavior of Π_1 and Π_2 on a finite set of design goals G . This suggests relativizing the partial order \preceq to a set G of goals of interest.

Definition 4 A protocol Π_2 is *at least as strong as* protocol Π_1 *under* G , written $\Pi_1 \preceq^G \Pi_2$, iff for every goal $\Gamma \in G$, if Π_1 achieves Γ then so does Π_2 .

When $\Pi_1 \preceq^G \Pi_2$ and $\Pi_2 \preceq^G \Pi_1$ we write $\Pi_1 \bowtie^G \Pi_2$.

When $\Pi_1 \preceq^G \Pi_2$ and $\Pi_2 \not\preceq^G \Pi_1$ we write $\Pi_1 \triangleleft^G \Pi_2$.

We want to be clever in choosing sets of goals G so that (1) it is straightforward to verify $\Pi_1 \preceq^G \Pi_2$, and (2) \preceq^G usefully distinguishes as many protocols as possible. The first condition pushes us to consider smaller sets, while the second condition pushes us to consider larger sets. In fact:

Lemma 2 $\Pi_1 \bowtie^G \Pi_2$ and $G' \subseteq G$ implies $\Pi_1 \bowtie^{G'} \Pi_2$.

3.4 Comparing Protocols by Finite Sets

The smallest useful set is a singleton $G = \{\Gamma\}$. In this case the induced partial order $\preceq^{\{\Gamma\}}$ is actually a two-point total order on protocols. Each protocol either achieves Γ or not. If we let $\Gamma_{(4)}$ be the PKINIT authentication goal in formula (4), then

$$\text{PKINIT} \triangleleft^{\{\Gamma_{(4)}\}} \text{PKINIT}_1 \bowtie^{\{\Gamma_{(4)}\}} \text{PKINIT}_2,$$

because PKINIT does not achieve the goal but both fixes do. Similarly, for the non-disclosure goal $\Gamma_{(7)}$ in formula (7),

$$\text{PKINIT} \triangleleft^{\{\Gamma_{(7)}\}} \text{PKINIT}_1 \bowtie^{\{\Gamma_{(7)}\}} \text{PKINIT}_2.$$

Thus, the two fixes agree on $\{\Gamma_{(4)}, \Gamma_{(7)}\}$, i.e.

$$\text{PKINIT}_1 \bowtie^{\{\Gamma_{(4)}, \Gamma_{(7)}\}} \text{PKINIT}_2.$$

Larger finite sets G may induce more interesting partial orders. When G represents the set of desired goals, it is the job of the protocol designer to define a protocol that achieves all the goals in G so that $G \subseteq \text{Ach}(\Pi)$. As standards committees propose fixes to protocols, however, cases may arise in which two proposals Π_1 and Π_2 are incomparable via \preceq^G . Such cases provide an opportunity for standards committees to discuss the relative importance of goals in G . It may be worthwhile to sacrifice some goals in order to achieve others. Recognizing that two protocols are incomparable in the partial order defined by the goals allows the trade-offs to be much clearer.

4 Related Work

There are several approaches to measuring the security of systems. One approach focuses on establishing a set of standardized names for common concepts, languages for expressing and sharing information about

those concepts, and system administrator tools to automatically collect measurements of a system with results expressible in those languages. Works exemplified by Martin [28], Sun et al. [36], and Liu et al. [25] demonstrate ways to leverage common enumerations such as CVE [12] and CWE [13] in automatically generating security metrics for the analysis of operational systems. Our approach differs in several crucial aspects. By focusing on the relatively narrow domain of cryptographic protocols, we can more easily develop a general and formal language of events in which to express security properties. We are thus able to provide a clear semantics for formulas in our language that corresponds to the effects of attacks without enumerating those attacks explicitly.

4.1 Protocol Security Hierarchies

More closely related to our work are approaches to analyzing security protocols relative to specific definitions of protocol security. There have been many attempts in the literature to define protocol security [6, 38, 34, 26, 9, 10, 1] and there has been no consensus on a formal definition of what is meant by that term. This is for good reason: Security comes in various forms and strengths. Some varieties may be suitable for one purpose, but insufficient for another.

This point was clearly made by Lowe in [26] when he defined a hierarchy of authentication specifications ranging from aliveness to injective agreement on a set of values. Since then, Cremers and Mauw [14] have amended and extended this hierarchy to capture synchronization properties (reminiscent of properties defined in Woo and Lam’s [38] and Roscoe’s [34]) and whether or not some party’s peer is running the expected role. In [4], Basin and Cremers identify a hierarchy of adversary models and derive a hierarchy of protocols according to the strongest adversary model under which the protocols are secure. Their adversary models come mostly from Canetti and Krawczyk’s [9, 10]. Security in this case is with respect to a small, fixed set of well-defined secrecy and authentication goals.

We view the present work as a step in the direction of unifying, simplifying, and extending the related work on hierarchies of specifications, adversary models, and protocols. Our security goal language is designed to express properties in a manner that is independent of the underlying formalisms or tools used to verify them. The structure of goals as first order formulas makes the relative strength of security goals clear and immediate. The language is expressive enough to capture the natural notions of authentication and secrecy used by others. Judicious use of the atomic predicates $\text{Non}(v)$,

$\text{Unq}(v)$, and $\text{Preceq}(m, n)$ can also express subtle limits on the adversary’s ability to compromise both long-term and short-term data. These are the dimensions along which Basin and Cremers vary their adversary models in [4]. Thus we can also incorporate a variety of adversary models into the specifications themselves, which Basin and Cremers identify as an alternative approach to theirs.

Our aim in the rest of this section is to elaborate the detailed connection with the related work on protocol security hierarchies. By recreating the authentication hierarchy as defined by Lowe [26] and extended by Cremers and Mauw [14], we hope the reader will gain a stronger intuition for how to uniformly express a wide variety of important security goals drawn from the literature.

4.2 Recreating a Hierarchy of Authentication Goals

Lowe begins his investigation by defining four types of authentication, *weak aliveness*, *weak agreement*, *non-injective agreement*, and *injective agreement*.¹ They are all stated from the perspective of an agent A playing the initiator role of a protocol trying to authenticate another agent B playing the responder role. They each assert the existence of some protocol event given that some behavior has occurred. There is a clear ordering of strength among them because they demand increasingly more agreement between A and B . Lowe notes that the restriction to two party protocols and to authentication of a responder by an initiator is incidental. The definitions naturally generalize to reversing the order of authentication and to multi-party protocols.

We now express each of these properties in our goal language. Although our language is independent of the particular protocol being considered, it does require role position predicates for each node and parameter predicates for the values at those nodes. Thus the actual formalization of these authentication goals will depend to some degree on the protocol. In order to remain as general as possible, we assume a protocol which has an initiator role and a responder role. The role position predicates $\text{IStart}(\cdot)$, $\text{RStart}(\cdot)$ will be satisfied by the first nodes of the initiator and responder roles respectively. Similarly $\text{IDone}(\cdot)$ and $\text{RDone}(\cdot)$ will be satisfied by their final nodes. We assume that, at the start of each role, parameters for its own identity and that of its peer are well defined. We use the parameter predicates $\text{Self}(\cdot, \cdot)$ and $\text{Peer}(\cdot, \cdot)$ to represent these parameters.

¹ We use the terminology of Cremers and Mauw’s [14] instead of [26] because it makes finer distinctions that are useful for our purposes.

All other parameters p are represented by predicates $\text{Param}_p(\cdot, \cdot)$.

Most protocol goals are trivially broken when the participants use compromised long-term keys or credentials. Exactly which keys must be kept secure to achieve certain goals will depend on the protocol. These assumptions about uncompromised keys can be expressed in our logic by naming the keys with $\text{Param}_k(n, v_k)$ and asserting they are unavailable to the adversary with $\text{Non}(v_k)$. We may use the $\text{Self}(n, A)$ or $\text{Peer}(n, A)$ parameter predicate with sk in $\text{Non}(\text{sk}(A))$. We use the notation $\text{GoodKeys}(n, \bar{k})$ to represent a conjunction of such formulas, expressing that the relevant keys (represented by the parameters \bar{k}) are not compromised. Our goals are thus somewhat parametric in which keys are covered in this formula.

The stronger authentication goals can naturally be expressed by modifying the weaker ones. To avoid typesetting large formulas that are difficult to read, we will define new formulas in terms of the parts of previous ones. Each goal has the form shown in Definition 1. We use the convention that for goal Γ_i the conjunction on the left of the implication is denoted by Φ_i . Each of the disjuncts on the right of the implication is denoted by Ψ_i^j for j between 1 and the number of disjuncts. Ψ_i^j includes the existential quantifier. Thus, authentication goal Γ_i is expressible as $\Phi_i \Rightarrow \bigvee_{1 \leq j \leq n} \Psi_i^j$ where any remaining free variables are implicitly universally quantified at the start of the formula. We build up larger formulas by adding conjuncts to these parts and capturing new variables under new existential quantifiers when needed.

Expressing the goals. We start by expressing the goals in Lowe’s hierarchy in our formalism. For the following discussion, we direct the reader to Table 2 which summarizes the results. Weak aliveness, Γ_1 , is the perhaps the weakest meaningful form of entity authentication. A protocol that satisfies weak aliveness guarantees that the intended peer started the protocol at some time in the past. It does not guarantee that the peer agrees on the initiator’s identity, nor does it guarantee that the peer is acting in the right role. The next property Γ_2 , weak agreement, specifies that the peer must also agree on the initiator’s identity. This is done by adding the relevant parameter predicates to the hypothesis Φ_1 and to each of the disjuncts Ψ_1^i .

Non-injective agreement requires the peer to be acting in the correct role (i.e. as a responder) and it requires that the two parties agree on some subset V of parameters used in their roles. We express non-injective agreement as Γ_3 which we obtain from Γ_2 in two steps. First we remove the disjunct Ψ_2^2 from the conclusion that allows the peer to act as an initiator. Then, for

each $p \in V$, we conjoin the corresponding parameter predicate $\text{Param}_p(\cdot, \cdot)$ to both the hypothesis and conclusion, ensuring the variables in the second argument are the same in both places.

The injective-agreement property is designed to ensure that a protocol is resistant to replay attacks in which an adversary may record messages from a previous successful session and use them at a later time. This can be avoided if the protocol ensures that each set of values the initiator commits to is unique to the current session. This *injective session property* is formalized as Γ_* . Injective agreement, Γ_4 , is then the conjunction of Γ_3 with Γ_* . It is possible to express injective agreement as a single goal, Γ_4' , but we believe it is more informative to demonstrate the logical independence of agreement and injectivity.

Lowe points out in [26] that properties Γ_1 through Γ_4 do not capture the notion of recentness. In many protocols the mechanisms to ensure the peer has been recently active are similar to those used to ensure the injective session property. Namely, random challenge nonces are used to ensure both that each session has some unique input and that the peer’s activities have occurred after the creation of the nonce. However, recentness and injectivity are logically independent. One is about the relative ordering of events, while the other is about the uniqueness of sessions. For each of the properties defined so far, Lowe defines another version that ensures recentness.

Recentness requires the existence of an event that is known to have occurred not too long ago. This could be a previous event performed by the initiator, or it could be an external event such as a clock tick. We remain agnostic about which event is used as a time reference. We simply assume we know what event in the protocol is sufficient for this purpose and we say that such a node satisfies the role position predicate $\text{TimeRef}(\cdot)$. For each Γ_i we obtain a version that requires recentness by modifying each of its disjuncts Ψ_i^j , adding the two conjuncts $\text{TimeRef}(m') \wedge \text{Preceq}(m', m)$. For example, we modify Γ_1 , weak aliveness, into Γ_5 , recent weak aliveness. Modifying Γ_2, Γ_3 and Γ_4 analogously, yields respectively

- Γ_6 , recent weak agreement;
- Γ_7 , recent non-injective agreement; and
- Γ_8 , recent injective agreement.

We omit these from Table 2 since they are completely analogous to Γ_5 .

In [14], Cremers and Mauw augment this hierarchy in two ways, which we formalize in Table 3. First, they choose to modify Γ_1 not by ensuring agreement on both identities, but by first ensuring that the peer is acting

Weak aliveness.

$$\left(\begin{array}{c} \text{IDone}(n) \wedge \text{Peer}(n, r) \wedge \\ \text{GoodKeys}(n, k) \end{array} \right) \implies \left(\begin{array}{c} (\exists m. \text{RStart}(m) \wedge \text{Self}(m, r)) \vee \\ (\exists m. \text{IStart}(m) \wedge \text{Self}(m, r)) \end{array} \right) \quad (\Gamma_1)$$

Weak agreement.

$$\Phi_1 \wedge \text{Self}(n, i) \implies (\Psi_1^1 \wedge \text{Peer}(m, i)) \vee (\Psi_1^2 \wedge \text{Peer}(m, i)) \quad (\Gamma_2)$$

Weak agreement: Variant.

$$\Phi_1 \implies \left(\begin{array}{c} (\exists i. \Psi_1^1 \wedge \text{Self}(n, i) \wedge \text{Peer}(m, i)) \vee \\ (\exists i. \Psi_1^2 \wedge \text{Self}(n, i) \wedge \text{Peer}(m, i)) \end{array} \right) \quad (\Gamma'_2)$$

Non-injective agreement.

$$\Phi_2 \wedge \bigwedge_{p \in V} \text{Param}_p(n, v_p) \implies \Psi_2^1 \wedge \bigwedge_{p \in V} \text{Param}_p(m, v_p) \quad (\Gamma_3)$$

Injective session.

$$\left(\begin{array}{c} \text{IDone}(n_1) \wedge \bigwedge_{p \in P(\text{init})} \text{Param}_p(n_1, v_p) \wedge \\ \text{IDone}(n_2) \wedge \bigwedge_{p \in P(\text{init})} \text{Param}_p(n_2, v_p) \end{array} \right) \implies n_1 = n_2 \quad (\Gamma_*)$$

Injective agreement.

$$\Gamma_3 \wedge \Gamma_* \quad (\Gamma_4)$$

Injective agreement: Variant.

$$\left(\begin{array}{c} \Phi_3 \wedge \text{IDone}(n') \wedge \\ \bigwedge_{p \in P(\text{init})} \text{Param}_p(n', v_p) \end{array} \right) \implies \Psi_3^1 \wedge n = n' \quad (\Gamma'_4)$$

Recent weak aliveness.

$$\Phi_1 \implies \left(\begin{array}{c} (\exists m'. \Psi_1^1 \wedge \text{TimeRef}(m') \wedge \text{Preceq}(m', m)) \vee \\ (\exists m'. \Psi_1^2 \wedge \text{TimeRef}(m') \wedge \text{Preceq}(m', m)) \end{array} \right) \quad (\Gamma_5)$$

Table 2 Authentication formulas in Lowe's hierarchy

Weak aliveness in role

$$\Phi_1 \implies \Psi_1^1 \quad (\Gamma_9)$$

Non-injective synchronization.

$$\Phi_3 \implies \exists \bar{m}. \Psi_3 \wedge \text{NodeOrder}(\bar{m}, n) \quad (\Gamma_{11})$$

Injective synchronization.

$$\Gamma_{11} \wedge \Gamma_* \quad (\Gamma_{12})$$

Table 3 Additional Cremers and Mauw authentication formulas

in the right role regardless of whether or not the peer correctly knows the initiator's identity. The result is weak aliveness in a role, Γ_9 , obtained by omitting Ψ_1^2 from Γ_1 . They also modify Γ_9 to obtain

Γ_{10} , yielding recent weak aliveness in a role, by adding $\text{TimeRef}(m') \wedge \text{Preceq}(m', m)$ to the conclusion.

Cremers and Mauw also introduce the notion of *synchronization* which has both a non-injective and injective variety. Non-injective synchronization is to be similar to the Bellare-Rogaway notion of matching conversations [6] and Roscoe's intensional security [34]. It

completely describes the intended protocol execution given completed run of the initiator. It requires that every transmission by the initiator precedes a matching reception by the peer and vice versa, where each corresponding pair of events agrees on the value of the message.

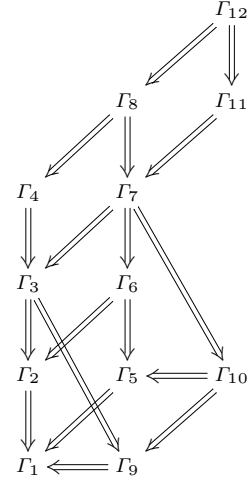
$\mathcal{GL}(\Pi)$ does not talk directly about this full message value. Instead, it talks about the parameters one-by-one. This limited expressiveness is actually a design target of our language: a security goal that depends too heavily on the exact structure of messages cannot be preserved under small syntactic changes. Since our language is designed for cross-protocol comparisons we have traded off expressibility for simultaneous applicability of goals to several protocols. In many cases, equalities between protocol messages can be enforced by stating that corresponding parameters of sender and receiver are equal. However, when they destructure the messages in different ways, this may not suffice. In a protocol such as PKINIT in which one party transmits a ciphertext it has encrypted, which another receives as an opaque blob, we cannot explicitly state that this component is unchanged.

Nonetheless, in many cases the specification of all the parameters will uniquely determine the message structure, and hence agreement on all parameters will entail agreement on the messages. In such cases we can formalize non-injective synchronization by modifying Γ_3 in the following way. We express the existence of each node we expect by the corresponding role position predicate. We express the desired ordering of these events using the $\text{Preceq}(\cdot, \cdot)$ and $\text{Coll}(\cdot, \cdot)$ predicates. This conjunction of predicates may be abbreviated by $\text{NodeOrder}(\bar{m}, n)$, where n is the only node variable in the hypothesis of Γ_3 . We then conjoin this to the conclusion Ψ_3 , existentially quantifying over all new node variables \bar{m} , resulting in non-injective synchronization, Γ_{11} . We do not add any more role parameter predicates or equalities because we assume the set V of parameters to agree on for Γ_3 is already the entire set of parameters $P(\text{init})$ and is enough to enforce agreement on the value of each message of the protocol.

To express injective synchronization we simply conjoin the injective session property to non-injective synchronization resulting in Γ_{12} .

Ordering the goals by strength. Having expressed the various goals in these hierarchies in our logical goal language, we now demonstrate their relative strength by implication. As we saw above, many goals are obtained from others by a handful of reusable modifications. Most of these modifications involve either adding conjuncts to the conclusion of an implication or removing disjuncts. Such modifications strengthen the implication by strengthening its conclusion. The rest of this section discusses this set of reusable modifications to goals and demonstrates that the resulting goals are stronger. The results are summarized in Fig. 4 where the implications on opposite sides of each of the parallelograms hold for analogous reasons.

We first work our way up the diagram vertically. In order to see that Γ_2 is stronger than Γ_1 we first note that we have obtained Γ_2 , in part, by adding conjuncts to the conclusion. If this were the only modification, it would immediately follow that Γ_2 was stronger than Γ_1 . However, we have also seemingly strengthened the hypothesis by adding the conjunct $\text{Self}(n, i)$, thereby potentially weakening the implication. In fact, however, it does not actually weaken the goal. We know that $\Phi_1 \Leftrightarrow \Phi_1 \wedge \exists i. \text{Self}(n, i)$ because we have assumed that the self parameter is always well defined at the first (and hence also last) node of the roles. Any skeleton satisfying $\text{IDone}(n)$ has a well-defined value for the self parameter at that node, and hence also satisfies $\text{Self}(n, i)$. Because of this we could have expressed Γ_2 as the alternate form Γ'_2 in Table 2, making it syntacti-



Γ_1 : Weak Aliveness	Γ_7 : Rec. non-inj. Agrmt.
Γ_2 : Weak Agrmt.	Γ_8 : Rec. inj. Agrmt.
Γ_3 : Non-inj. Agreement	Γ_9 : Weak Aliveness in Role
Γ_4 : Inj. Agreement	Γ_{10} : Rec. Aliveness in Role
Γ_5 : Rec. Weak Aliveness	Γ_{11} : Non-inj. Synch
Γ_6 : Rec. Weak Agrmt.	Γ_{12} : Inj. Synch

Fig. 4 Combined hierarchy

cally evident that it is stronger than Γ_1 . Γ_6 is stronger than Γ_5 for the same reasons.

To see that Γ_3 is a stronger requirement than Γ_2 we argue similarly. The consequent is strengthened by removing one of the disjuncts and adding a conjunct to the remainder. Again, although it looks like a strengthening of the antecedent, the set V is chosen to ensure that for any $p \in V$, $\text{IDone}(n) \Rightarrow \exists v_p. \text{Param}_p(n, v_p)$. In order for non-injective agreement to be satisfiable, it may be necessary to replace $\text{RStart}(m)$ with a later node such as $\text{RDone}(m)$. The issue is that the responder may not have learned or committed to all the relevant values in V at its first node. Such a modification only serves to further strengthen the goal because later nodes always imply the existence of earlier nodes of the same strand. Similarly, we conclude that Γ_7 is stronger than Γ_6 .

Injective agreement Γ_4 is clearly stronger than non-injective agreement Γ_3 since Γ_4 simply requires an additional goal to be met. For the same reason, we can conclude that $\Gamma_8 \Rightarrow \Gamma_7$ and that $\Gamma_{12} \Rightarrow \Gamma_{11}$.

Γ_9 strengthens Γ_1 since it results from omitting one of the disjuncts of Γ_1 's conclusion. For the same reason $\Gamma_{10} \Rightarrow \Gamma_5$.

Γ_5 is obtained from Γ_1 by adding the requirement for recency. This is done by adding conjuncts to the conclusion, thereby strengthening the goal. The same rea-

soning justifies all the parallel implications that simply add recency to a goal.

To see that $\Gamma_3 \Rightarrow \Gamma_9$, we recognize that both goals require the peer to be in the expected role, but Γ_3 requires more agreement among the parameters. That is, Γ_3 can be obtained from Γ_9 by adding parameter predicates to both the hypothesis and the conclusion. As we argued above, the addition of these predicates to the hypothesis does not strictly strengthen the hypothesis. Thus the additional parameter predicates conjoined to the conclusion can only strengthen the goal. Γ_{10} similarly strengthens Γ_5 .

Finally, we argue that Γ_{11} is stronger than Γ_7 . The conjunction $\text{NodeOrder}(\bar{m}, n)$ is typically stronger than the conjunction $\text{TimeRef}(m') \wedge \text{Preceq}(m', m)$, because $\text{TimeRef}(m')$ is usually one of the role position predicates included in $\text{NodeOrder}(\bar{m}, n)$, and $\text{Preceq}(m', m)$ is one of the required orderings. Thus $\text{NodeOrder}(\bar{m}, n)$ asserts the existence of more nodes and more orderings among them. This means that synchronization is a stronger requirement than recency. Thus we see that $\Gamma_{11} \Rightarrow \Gamma_7$, and similarly, $\Gamma_{12} \Rightarrow \Gamma_8$.

5 Protocols and Policies

One common source of resistance to the use of formal verification tools is their inability to capture the wide variety of realistic assumptions about the environment in which a protocol operates. When a potential attack is presented to a standards committee, it is common for the committee to argue that the attack violates some assumption guaranteed externally by some policy. In Section 5.1 we use an example protocol from an ISO standard [24] to show how we can use our goal language to capture not only properties that a protocol must enforce, but also limitations on the adversary that are enforceable through policy decisions. In Section 5.2 we discuss how a well-known flaw in the Transport Layer Security (TLS) protocol, discovered in 2009 [33], illuminates the tension that may arise when applications rely on a protocol to provide enough information to make local policy decisions. We show how our goal language might help structure standards bodies' understanding of the root cause of such a flaw and appropriate ways to mitigate it.

5.1 Policy-Based Constraints

The hierarchy is rather extensive, and yet it is still insufficient to express many goals or properties that arise for protocols developed as standards. In [3], Basin et al. use Scyther to analyze the ISO/IEC 9798 standard for

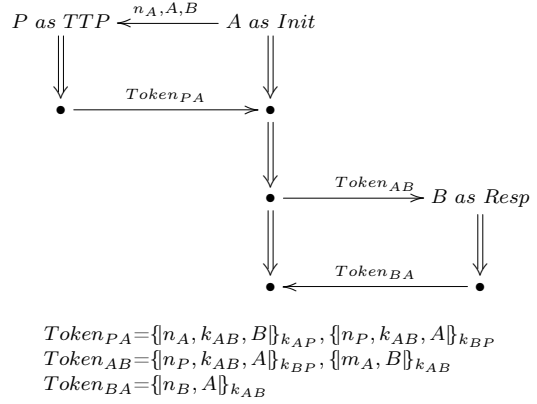


Fig. 5 ISO/IEC 9798-2, Mechanism 5

authentication protocols. They measure the protocols against goals in the above hierarchy and discover numerous cases in which the protocols do not achieve the goals. They then propose repairs to the protocols and demonstrate that the repaired protocols each achieve at least recent non-injective agreement. Interestingly, their alterations to the protocol messages were not sufficient to repair all the protocols. The repairs to two mechanisms from Part 2 of the standard [24] only achieves the goal under the extra external assumption that some policy is enforced. We dedicate the remainder of this section to demonstrating how we can apply our goal language to one of these protocols to express the adequacy of the protocol repairs under such policy-based constraints.

Figure 5 depicts the expected execution the protocol identified as Mechanism 5 in [24]. Two parties, A and B , want to authenticate each other using an on-line trusted third party (TTP) P . We assume that P shares symmetric keys k_{AP} and k_{BP} with A and B respectively. A , acting as initiator, begins by sending a nonce n_A together with B 's identity to P . P , acting in the role of TTP, creates a fresh, symmetric session key k_{AB} to be used between A and B . The key is encrypted separately for A and B together with the random values n_A and n_P and the identities of the intended peers. A passes along P 's encryption for B together with another encryption using the session key k_{AB} of a random value m_A and B 's identity. The responder B completes the protocol by using k_{AB} to encrypt a random value n_B and A 's identity.

The protocol as described does not achieve non-injective agreement because many of the encryptions have similar structure. This allows an adversary to combine messages from two sessions in which A and B are both acting as initiator to convince A that B is acting

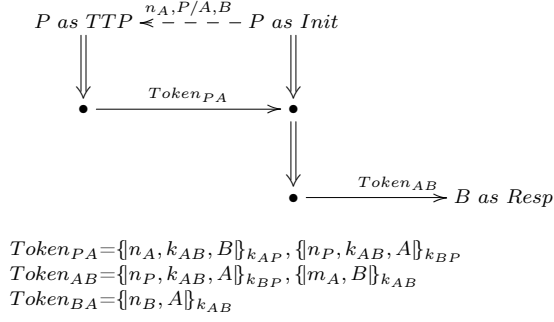


Fig. 6 Attack on ISO/IEC 9798-2, Mechanism 5

as responder. Basin et al. suggest including unique tags in each of the encryptions so that such role confusions are no longer possible. Although this change eliminates the role-mixup attacks, the resulting protocol still does not achieve non-injective agreement (in fact, it does not even achieve weak aliveness).

Figure 6 displays a remaining attack on the protocol. In this instance P is executing both the initiator and the TTP roles simultaneously. As initiator, P sends the nonce and the two identities P and B . The adversary swaps P 's identity with A 's identity before delivering the message to P in the TTP role (represented with P/A over the dotted line). As TTP, P believes the request to be coming from A wanting to talk with B , so he prepares encryptions for both A and B using the long-term shared symmetric keys k_{AP} and k_{BP} . The adversary can redirect this message back to P in the initiator role. P is willing to accept this message because it believes it to be coming from A acting as TTP and it is encrypted with the key k_{AP} that he shares with A . When P acting as initiator sends the final message to B , B believes that A was active at some point which is not true. This violates weak aliveness of A from the responder's point of view.

The root cause of this problem is that the second message does not contain enough information about who the TTP thinks is playing which role, assuming that a principal can play both the TTP role and the initiator role. This suggests two natural ways to address the attack. The first is to alter the message structure to add more detailed information about which principals appear to be playing which roles. The second is to eliminate the ambiguity by forbidding any principals that acts as TTP to also act as either initiator or responder. While the first option may be desirable because it allows for more flexible deployment scenarios, it may be difficult for a standards committee to accept such a drastic change to the message structure when numerous implementations exist. Furthermore, the first op-

tion requires the protocol implementors to create and distribute a patch, while the second option empowers any enterprise that is an end user of the protocol to protect themselves with a policy change.

Our goal language makes it straightforward to evaluate whether a particular proposal to amend the protocol will achieve the desired authentication goal as we demonstrated for the case of PKINIT. But how can we use our techniques to evaluate whether a change in local policy will allow the protocol to achieve its goal? An analysis of the protocol will always tell us that it does not even achieve weak aliveness. We must capture what the protocol does achieve and ensure that it entails a goal that says either the protocol violates some external policy or else it achieves non-injective agreement.

The external policy can be viewed as an extra assumption we would like to make on our analysis. While goals typically represent assumptions as conjuncts in the antecedent, our language restricts us to using positive statements such as the fact that some event has occurred, or some value is freshly chosen. This policy is really a negative statement: A principal will not play both the TTP role and another role. We can express such requirements by negating the policy to make a positive statement and place that in the disjunctions of the conclusion. So if we let $\Phi'_3 \Rightarrow \Psi'_3$ represent the goal of non-injective agreement with the initiator from the responder's point of view, then we can use the following goal:

$$\Phi'_3 \implies \Psi'_3 \vee \Psi_a \vee \Psi_b \quad (\Gamma_{13})$$

in which we let

$$\begin{aligned} \Psi_a &= \exists n_1, n_2, v. \text{ TTPStart}(n_1) \wedge \text{Self}(n_1, v) \wedge \\ &\quad \text{InitStart}(n_2) \wedge \text{Self}(n_2, v) \\ \Psi_b &= \exists n_1, n_2, v. \text{ TTPStart}(n_1) \wedge \text{Self}(n_1, v) \wedge \\ &\quad \text{RespStart}(n_2) \wedge \text{Self}(n_2, v). \end{aligned}$$

The disjunction $\Psi_a \vee \Psi_b$ captures a violation of the policy that restricts any principal that plays the TTP role from playing either the initiator role (Ψ_a) or the responder role (Ψ_b). Using CPSA we have verified that the protocol achieves goal Γ_{13} thereby demonstrating that such a policy, if properly enforced, can ensure the protocol achieves non-injective agreement.

Notice that Γ_{13} fits only tangentially into the hierarchy of Fig. 4. It is weaker than non-injective agreement because we achieved it by weakening the conclusion, adding extra disjuncts. However, it is not stronger than any of the other goals in the hierarchy. This example demonstrates the inadequacy of such hierarchies in addressing the real goals and constraints that arise

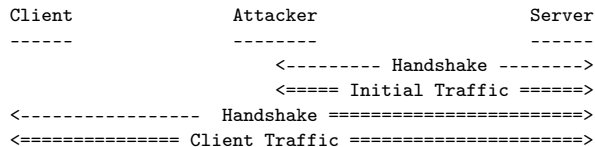


Fig. 7 TLS renegotiation attack

in the standardization of security protocols. We believe that combining our goal language with formal verification tools goes a long way to offering the flexibility and extensibility necessary to naturally express and verify goals and policy-based constraints that arise during the development and maintenance of protocol standards.

5.2 TLS Renegotiation

The policy in Section 5.1 is encoded in our security goal Γ_{13} . One aspect of that example that made it particularly clean is that the policy is a global one that can be enforced during the registration of participants. That is not always the case. Some policy decisions may dynamically rely on information provided by the protocol itself, for example when an application is relying on the underlying protocol for certain guarantees. When a surprising behavior—potentially a flaw—is discovered, we may need to change what information is available at the interface between the protocol and the application. The application can then use this information to correctly enforce a policy. We start with an example.

Transport Layer Security (TLS) [15] is a globally deployed protocol designed to add confidentiality, authentication and data integrity between communicating applications. It is secure, scalable, and robust enough to protect e-commerce transactions over HTTP. Despite its success, it has been forced to evolve over time, in part due to the discovery of various flaws in the design logic.

One such flaw, discovered in 2009 by Marsh Ray (see [33]), concerns renegotiating TLS parameters. It works on the boundary between the TLS layer and the application layer it supports. Fig. 7 is a high-level picture of the attack borrowed from [33]. The attacker first creates a *unilaterally* authenticated session with the server in the first handshake. Thus, the server authenticates itself to the attacker, but not vice versa. The attacker and server then exchange initial traffic protected by this TLS session. Later, a renegotiation occurs, possibly when the application at the server requires *mutual* authentication for some action. The attacker then allows the client to complete a handshake with the server, adding and removing TLS protections. The client’s handshake occurs in the clear (depicted by

<----> in Fig. 7), while the server’s handshake is protected by the current TLS session. The attacker has no access to this newly negotiated session, but the server may retroactively attribute data sent in the previous session to the authenticated client. The server may then perform a sensitive action in response to a request sent by the attacker, but based on the credentials subsequently provided by the client.

In a typical illustration, the server is a pizza shop. The attacker orders a pizza for delivery to his address; then the client authenticates and orders a pizza for his own address. Because this is (now) a bilaterally authenticated connection, the server charges the orders made in this process to the client’s account.

Which level is to blame for this attack?

- Does TLS fail to achieve a security goal that it should achieve?
- Or should the application take responsibility? It accepts some data out of a stream that is not bilaterally authenticated, and lumps it with the future data which will be bilaterally authenticated.
- Or is there shared responsibility? Perhaps TLS has the responsibility of providing clearer indications to the application when a change in the TLS properties takes place, and the application has the responsibility of heeding these indications.

TLS was subsequently updated with a renegotiation extension [33]. Renegotiation now creates a cryptographic binding between the new session and the existing session. If a server completes a mutually authenticated renegotiation with a client, then the earlier session was also negotiated with the same client. However, the authors of [33] also note:

While this extension mitigates the man-in-the-middle attack described in the overview, it does not resolve all possible problems an application may face if it is unaware of renegotiation.

As Bhargavan et al. [7]’s recent attacks showed, the practically important issue was not in fact resolved by this. Perhaps future versions of TLS will simplify the situation by eliminating renegotiation.

However, the main point stands: for applications to use a security protocol such as TLS effectively, and take partial responsibility for achieving reasonable policies, some signals and commands must be shared between TLS and the application. Formal verification—coupled with our goal language—fits naturally here. With a little effort the goal language can be updated to address the multilayer nature of flaws such as this.

5.3 Enforcing Policies at Protocol Interfaces

The job of TLS, acting in either direction, is to take a stream of data from an application, delivering as much as possible of this stream to the peer application. When the sender is authenticated to the receiver, TLS guarantees that the portion delivered is an initial segment of what the authenticated sender transmitted. When the mode offers confidentiality, no other principal should learn about the content (beyond its length).

Naturally, these goals are subject to the usual assumptions, such as that the certificate authorities are trustworthy, that the private keys are uncompromised, and that randomness was freshly chosen.

When renegotiation occurs, this affects what the application should rely on. If a handshake authenticates a client identity C , then that guarantee should apply to the data starting when the cipher spec changes. It continues to apply until another cipher spec change or the end of the connection. Authentication guarantees for the “client traffic” should definitely not apply to the “initial traffic” of Fig. 7, which lies on the other side of a cipher spec change.

We may regard the TLS record layer as engaging in two types of events, namely the network-facing events and the application-facing events. Consider the outbound stream of data from the application to its peer. The network-facing events are transmission events for messages on the network to the peer. The application-facing events *accept* a stretch of data from the application, to be transmitted to the peer via the network. These stretches of data may be of various lengths, so that the record layer has to break them into many TLS records before network transmission.

For the inbound stream of data from the peer, the network-facing events are reception events for messages on the network from the peer. The application-facing events *deliver* a stretch of data to the application. As the record layer assembles these stretches of data from many TLS records on the network, they may be of various lengths.

Thus, in representing the TLS record layer, we use four kinds of node:

NetReceive(n): n is a network-facing event receiving a record;

NetSend(n): n is a network-facing event transmitting a record;

AppAccept(n): n is an application-facing event accepting data from the application to go to the peer;

AppDeliver(n): n is an application-facing event delivering data from the peer to the application.

Several parameter predicates may apply to these nodes. If **NetReceive**(n) or **NetSend**(n), then n certainly has

a MAC key and an encryption key, as well as sequence number and payload data. We will not pause to formalize these, nor to define the roles that these events lie on. They are most easily formalized as protocol roles that interact with mutable state [21, 31].

When **AppAccept**(n) or **AppDeliver**(n), then three parameter predicates that apply to n are:

AppData(n, d): n is accepting or delivering the application data d ;

AppPeer(n, p): the authenticated peer at the time n occurs is p ;

AppSelf(n, p): the authenticated identity of the active principal at the time n occurs is p .

The predicates **AppPeer**(n, p) and **AppSelf**(n, p) may not hold of any value p if the endpoint is unauthenticated when n occurs.

Using these notions about the protocol behavior, the application can express its requirements about the security services that the protocol provides. We are interested in a situation where two segments of data are delivered to the pizza server, and it replies with a segment completing the transaction. We assume that the application defines a method to concatenate data segments d_1, d_2 , which we write $d_1 \hat{\ } d_2$; and that it has a way to assert that data d expresses a well-formed transaction **Transact**(d). Then the application is interested in the situation when:

$$\begin{aligned} & \mathbf{AppAccept}(n_1) \wedge \mathbf{AppAccept}(n_2) \\ & \wedge \mathbf{AppDeliver}(n_3) \wedge \\ & \bigwedge_{1 \leq i \leq 3} \mathbf{AppData}(n_i, d_i) \wedge \mathbf{Transact}(d_1 \hat{\ } d_2 \hat{\ } d_3). \end{aligned}$$

In this situation, we certainly want it to be the case that there is a single client c which is the peer of all three events:

$$\begin{aligned} & \exists c. \mathbf{AppPeer}(n_1, c) \wedge \mathbf{AppPeer}(n_2, c) \\ & \wedge \mathbf{AppPeer}(n_3, c). \end{aligned}$$

The authentication properties of the protocol itself may also imply further conclusions, such as c actually having transmitted $d_1 \hat{\ } d_2$.

Thus, the goal language we have provided can be enriched to express events at the interface between the protocol and its application. This notation—augmented with some application level vocabulary—allow the application author to express the meaningful conclusions that should hold whenever the application obtains its security services from the protocol.

6 Comparing Protocols by Hypotheses

Up to now, we have limited ourselves to considering only finite sets of goals G for the comparison of pro-

protocols. Using finite sets is helpful because $\text{Ach}(\Pi) \cap G$ can be enumerated. The limitation of considering only finite sets of goals is that the distinguishing power of the induced partial order \leq^G is limited by the size of G (cf. Lemma 2). We may lack the imagination to ensure G includes goals that would help distinguish protocols in a useful way. A more flexible approach might be to fix a set of assumptions and try to compare protocols according to the strength of the conclusions they allow. For example, Π_2 should be considered stronger than Π_1 if Π_2 allows an initiator to conclude more information about their peer’s session than Π_1 allows. This insight leads us to consider the following set of goals that all share a common hypothesis:

$$H(\Phi) = \{\Gamma \mid \text{hyp}(\Gamma) = \Phi\}. \quad (9)$$

Thus, for $\Gamma, \Gamma' \in H(\Phi)$, $\Gamma \leq \Gamma'$ iff $\text{conc}(\Gamma') \Rightarrow \text{conc}(\Gamma)$. $H(\Phi)$ is typically not a finite set, so we can no longer calculate $\text{Ach}(\Pi) \cap H(\Phi)$ by enumeration.

However, using our tool CPSA [32], we can find—for a large, natural class of assumptions Φ —a single strongest formula in $\text{Ach}(\Pi) \cap H(\Phi)$. We will call this formula the *shape analysis sentence* $\text{SAS}_\Phi(\Pi)$. It serves to summarize the whole set $\text{Ach}(\Pi) \cap H(\Phi)$. The key idea here is *enrich-by-need protocol analysis*. As we will see, enrich-by-need analysis outputs such a maximally strong formula.

Thus the sets $H(\Phi)$ are a natural companion to enrich-by-need protocol analysis. This raises the question of whether the analysis methods used by other tools have similar connections to other classes of goals.

6.1 Enrich-by-Need Protocol Analysis

Our approach to protocol analysis is based on what we call the “point-of-view principle.” Most of the security goals we care about in protocol design and analysis concern the point of view of a particular participant C . C *knows* that it has sent and received certain messages in a particular order. C may be willing to *assume* that certain keys are uncompromised, which for us means that they will be used only in accordance with the protocol in question. And C may also be willing to *assume* that certain randomly chosen values will not also be independently chosen by another participant, whether a regular (compliant) participant including C itself on another occasion, or an adversary.

These facts and assumptions may be formalized in a hypothesis. The hypothesis in Eqn. 3 is an example. It summarizes the situation in which a client has executed a full local run; it declares the variables C, S to stand for two of the parameters of the client run, and it adds

the assumption that the signature key $\text{sk}(S)$ of the peer is uncompromised.

The protocol analysis question is, given these facts and assumptions, what follows about what may happen on the network? The answers to this question are of two main kinds. Positive conclusions assert that some regular participant Q has taken protocol actions. These are authentication goals. They assert, subject to the assumptions, that C ’s message transmissions and receptions authenticate Q as having taken corresponding actions. Negative conclusions are generally non-disclosure assertions. They say that a value cannot be found available on the network in a particular form; often, that a key k cannot be observed unprotected by encryption on the network.

Skeletons and cohorts. The enrich-by-need process starts with a representation of the hypothesis. We will refer to these representations of behavior and assumptions as *skeletons* \mathbb{A} . A skeleton consists of some behavior of Π , i.e. some *regular strands* in the strand space terminology, together with the stated assumptions.

A *skeleton* \mathbb{A} for Π is a structure that provides partial information about a set of Π -bundles. It consists of a finite set of regular strands of Π , or initial segments of them; some assumptions about which keys should be assumed uncompromised and which should be assumed to have been freshly chosen; and a partial ordering on the nodes of Π . A bundle \mathcal{B} is an *instance* of a skeleton \mathbb{A} if there is a structure-preserving map (“homomorphism”) from the strands of \mathbb{A} into the regular strands of \mathcal{B} such that \mathcal{B} satisfies the freshness and non-compromise assumptions of \mathbb{A} , and its arrows enrich the partial ordering of \mathbb{A} (see [20] for copious detail).

CPSA [32] is a software tool that carries out protocol analysis in strand spaces. Given any skeleton \mathbb{A}_0 as a starting point, CPSA provides information about the set of all bundles \mathcal{B} such that there is a homomorphism from \mathbb{A}_0 into \mathcal{B} . CPSA does this by computing a set of skeletons $\{\mathbb{B}_i\}_{i \in I}$ with two properties. First, each of the skeletons \mathbb{B}_i is a *realized skeleton*. By this we mean that there is a bundle \mathcal{B}_i such that \mathbb{B}_i results from \mathcal{B}_i when we “forget” the specific adversary strands in \mathcal{B}_i . Second, *every* bundle \mathcal{B} containing an image of \mathbb{A}_0 also contains an image of one of the \mathbb{B}_i . We summarize these properties by saying that $\{\mathbb{B}_i\}_{i \in I}$ *characterizes* the skeletons compatible with the starting point \mathbb{A}_0 .

We start the enrich-by-need analysis with a skeleton \mathbb{A}_0 . At any point in the enrich-by-need process, we have a set \mathcal{S} of skeletons to work with. Initially, $\mathcal{S} = \{\mathbb{A}_0\}$.

At each step, we select one of these skeletons $\mathbb{A} \in \mathcal{S}$, and ask if the behavior of the participants recorded in it is possible. When a participant receives a message, then the adversary should be able to generate that message,

using messages that have been sent earlier, without violating the assumptions. In this case, we regard that reception as “explained,” since we know how the adversary can arrange to deliver the expected message. We say that that particular reception is *realized*. When every reception in a skeleton \mathbb{A} is realized, we call \mathbb{A} itself realized. It then represents—together with behavior that the adversary can supply—a possible complete execution, i.e. a bundle.

We collect the realized skeletons in a set \mathcal{R} .

If the skeleton $\mathbb{A} \in \mathcal{S}$ we select is not realized, then we use a small number of rules to generate an enrichment step. An enrichment step takes one unrealized reception and considers how to add some or all of the information that the adversary would need to generate its message. It returns a *cohort* of skeletons: This is a finite set $\{\mathbb{A}_1, \dots, \mathbb{A}_i\}$ of skeletons forming a disjunctive representation of all the ways the regular participants can supply the necessary information to the adversary. We update \mathcal{S} by removing \mathbb{A} and adding the cohort members: $\mathcal{S}' = (\mathcal{S} \setminus \{\mathbb{A}\}) \cup \{\mathbb{A}_1, \dots, \mathbb{A}_i\}$.

As a special case, a cohort may be the empty set, i.e. $i = 0$, and in this case \mathbb{A} is discarded and nothing replaces it. This occurs when there are no possible behaviors of the regular participants that would explain the required reception. Then the skeleton \mathbb{A} cannot contribute any executions (realized skeletons).

This process may not terminate, and in fact the underlying class of problems is undecidable [17]. However, when it does terminate, it yields a finite set \mathcal{R} of realized skeletons with a crucial property: For a class of security goals, if they have no counterexample in the set \mathcal{R} , then the protocol really achieves that goal [22]. Moreover, we can inspect the members of \mathcal{R} and determine whether any of them is a counterexample.

We call the members of \mathcal{R} *shapes*, and they represent the *minimal, essentially different* executions consistent with the starting point.

Enrich-by-need analysis originates with Meadows’s NPA [29]. Dawn Song’s Athena [35] applied the idea to strand spaces [37]. Two systems in use currently that use the enrich-by-need idea in a form close to what we describe here are Scyther [14] and our CPSA [32]. See [20, 22] for a comprehensive discussion, and for more information about our terminology here.

Example: Initiator’s authentication guarantee in PKINIT. Suppose that the client C has executed a strand of the client role in the fixed PKINIT₁, i.e., where we instantiate $F(C, n_2) = (C, n_2)$. Suppose also that we are willing to assume that the authentication server S has an uncompromised signature key $\text{sk}(S)$. We annotate this assumption as $\text{sk}(S) \in \text{non}$, meaning that $\text{sk}(S)$ is non-compromised.

$$\begin{array}{c} \text{sk}(S) \in \text{non} \quad \bullet \longrightarrow \bullet \\ \quad \quad \quad \downarrow s_1 \\ \quad \quad \quad \bullet \longleftarrow \bullet \end{array} \quad (10)$$

$s_1 \in \text{Client}[C, S, T, n_1, n_2, t_C, t_S, k, AK]$

This is our starting point \mathbb{A}_0 , depicted in (10), which corresponds to the information expressed in formula (3) from Section 2. C receives a message that contains the digital signature $[k, (C, n_2)]_{\text{sk}(S)}$, and we know that the adversary cannot produce this because $\text{sk}(S)$ is uncompromised. Thus, this second node of the local run is unrealized.

To explain this reception, we look at the protocol to see what ways a regular participant might create a message of the form $[k, (C, n_2)]_{\text{sk}(S)}$. In fact, there is only one. Namely, the second step of the KAS role does so. Knowing the KAS sends this signature means it will agree on the parameters used: S, k, C, n_2 . However, we do not yet know anything about the other parameters used in S ’s strand. They could be different values $t'_C, T', n'_1, TGT', AK', t'_S$. Thus, we obtain a cohort containing a single skeleton \mathbb{A}_1 , depicted in (11), that includes an additional KAS strand with the specified parameters.

$$\begin{array}{c} \text{sk}(S) \in \text{non} \quad \bullet \longrightarrow \longrightarrow \bullet \\ \quad \quad \quad \downarrow s_1 \quad \quad \quad \downarrow s_2 \\ \quad \quad \quad \bullet \longleftarrow \longleftarrow \bullet \end{array} \quad (11)$$

$s_1 \in \text{Client}[C, S, T, n_1, n_2, t_C, t_S, k, AK]$
 $s_2 \in \text{KAS}[C, S, T', n'_1, n_2, t'_C, t'_S, k, AK']$

This skeleton is now already realized, because, with this weak assumption, the adversary may be able to use C ’s private decryption key to obtain k and modify the authenticator $\{AK, n_1, t_S, T\}_k$ as desired. The adversary might also be able to guess k , e.g. if S uses a badly skewed random number generator. Similarly, the components that are not cryptographically protected are under the power of the adversary.

We can now re-start the analysis with two additional assumptions to eliminate these objections. First, we add $\text{sk}(C)$ to non . Second, we assume that S randomly generates k , and we write $k \in \text{unique}$, meaning that k is chosen at a unique position. We are uninterested in the negligible probability of a collision between k and a value chosen by another principal, even one chosen by the adversary. This situation is depicted in (12).

$$\begin{array}{c} \text{sk}(S), \text{sk}(C) \in \text{non} \quad \bullet \longrightarrow \longrightarrow \bullet \\ \quad \quad \quad \downarrow s_1 \quad \quad \quad \downarrow s_2 \\ \quad \quad \quad \bullet \longleftarrow \longleftarrow \bullet \\ k \in \text{unique} \end{array} \quad (12)$$

$s_1 \in \text{Client}[C, S, T, n_1, n_2, t_C, t_S, k, AK]$
 $s_2 \in \text{KAS}[C, S, T', n'_1, n_2, t'_C, t'_S, k, AK']$

This skeleton is not realized, because with the assumption on $\text{sk}(C)$, the adversary cannot create the signed unit $[t'_C, n_2]_{\text{sk}(C)}$; it must come from a compliant principal. Examining the protocol, this can only be the first node of a client strand with matching parameters C, n_2, t'_C , i.e. a local run s_0 :

$$\begin{array}{c}
 \text{sk}(S), \text{sk}(C) \in \text{non} \\
 k \in \text{unique} \\
 \begin{array}{ccc}
 \bullet & \longrightarrow & \bullet \\
 s_0 \downarrow & & s_1 \downarrow \\
 \circ & \longleftarrow & \bullet \\
 & & s_2 \downarrow
 \end{array}
 \end{array} \quad (13)$$

$$\begin{array}{l}
 s_0 \in \text{Client}[C, S'', T'', n_1'', n_2, t'_C, t'_S, k'', AK''] \\
 s_1 \in \text{Client}[C, S, T, n_1, n_2, t_C, t_S, k, AK] \\
 s_2 \in \text{KAS}[C, S, T', n_1', n_2, t'_C, t'_S, k, AK']
 \end{array}$$

Curiously, two possibilities now remain. The strand s_0 could be identical with s_1 , in which case the doubly-primed parameters equal the unprimed ones. Or alternatively, s_0 might be another client strand that has by chance selected the same n_2 ; we have not assumed $n_2 \in \text{unique}$. In this case, the doubly-primed variables are not constrained. If we further add the assumption that $n_2 \in \text{unique}$, then $s_1 = s_0$ follows.

In all of these cases, C and S do not have to agree on TGT , since this item is encrypted with a key shared between S and T , and C cannot decrypt it or check any properties about what he receives.

We have illustrated several points about enrich-by-need protocol analysis. Authentication follows by successive inferences about regular behavior, based on the message components the adversary cannot build. When two inferences are possible, the method branches, potentially resulting in several outputs. Various levels of authentication may be achieved; they depend on which parameters principals agree on, and which parameters may vary. Secrecy properties hold when no execution is compatible with the secret's disclosure.

More formally, there is a notion of homomorphism between skeletons [20]. Suppose that we start with the “point of view” expressed in the skeleton \mathbb{A}_0 , and CPSA terminates, providing us with the shapes $\mathbb{C}_1, \dots, \mathbb{C}_i$, which are the minimal, essentially different executions compatible with \mathbb{A}_0 . Then for each \mathbb{C}_j , there is a homomorphism H_j from \mathbb{A}_0 to \mathbb{C}_j . Moreover, every homomorphism $K: \mathbb{A}_0 \rightarrow \mathbb{D}$ from \mathbb{A}_0 to a realized skeleton \mathbb{D} agrees with at least one of the H_j . Specifically, we can regard K as the result of adding more information after one of the H_j . We mean that we can always find some $J: \mathbb{C}_j \rightarrow \mathbb{D}$ where K is the composition $K = J \circ H_j$.

6.2 Shape Analysis Formulas

Given a skeleton, we can summarize all of the information in it in the form of a conjunction of atomic formulas. We call this formula the *characteristic formula* for

the skeleton, and write $\text{cf}(\mathbb{A})$. Thus, a CPSA run with starting point \mathbb{A}_0 is essentially exploring the security consequences of $\text{cf}(\mathbb{A}_0)$.

When CPSA reports that \mathbb{A}_0 leads to the shapes $\mathbb{C}_1, \dots, \mathbb{C}_i$, it is telling us that any formula that is true in all of these skeletons, and is preserved by homomorphisms, is true in all realized skeletons \mathbb{D} accessible from \mathbb{A}_0 . The set of formulas preserved by homomorphism are called *positive existential*, and are those formulas built from atomic formulas, \wedge, \vee , and \exists . By contrast, formulas using negation $\neg\phi$, implication $\phi \implies \psi$, or universal quantification $\forall y. \phi$ are not always preserved by homomorphisms.

Thus, the disjunction of the characteristic formulas of the shapes $\mathbb{C}_1, \dots, \mathbb{C}_i$ tell us just what security goals \mathbb{A}_0 leads to. However, we can be somewhat more precise. The skeleton \mathbb{C}_j may have nodes that are not in the image of \mathbb{A}_0 , and it may involve parameters that were not relevant in \mathbb{A}_0 . Thus, \mathbb{A}_0 will not determine exactly which values these new items take in \mathbb{C}_j , e.g. which session key is chosen on some local run not present in \mathbb{A}_0 . Thus, these new values should be existentially quantified. Effectively, these are all the variables that do not appear in $\text{cf}(\mathbb{A}_0)$. Thus, for each \mathbb{C}_j , let \bar{y}_j list all the variables in $\text{cf}(\mathbb{C}_j)$ that are not in $\text{cf}(\mathbb{A}_0)$. Let \bar{x} list all the variables in $\text{cf}(\mathbb{A}_0)$. Then this run of CPSA has validated the following formula that has the required form of a security goal (Definition 1):

$$\forall \bar{x}. (\text{cf}(\mathbb{A}_0) \implies \bigvee_{1 \leq j \leq i} \exists \bar{y}_j. \text{cf}(\mathbb{C}_j)) \quad (14)$$

The conclusion $\bigvee_{1 \leq j \leq i} \exists \bar{y}_j. \text{cf}(\mathbb{C}_j)$ is the strongest formula that is true in all of the \mathbb{C}_j .

We call the formula (14) the *shape analysis sentence* for this run of CPSA. Since the empty disjunction is the constantly false sentence, in the special case where $i = 0$, (14) is $\forall \bar{x}. \text{cf}(\mathbb{A}_0) \implies \text{false}$. That is to say, it is $\forall \bar{x}. \neg \text{cf}(\mathbb{A}_0)$.

We will frequently have need to reference particular shape analysis sentences and their various parts. We introduce here a notation for ease of presentation. Suppose the characteristic formula $\text{cf}(\mathbb{A}_0)$ in protocol Π is represented by the conjunction of atomic formulas Φ , and each resulting shape is formalized by $\exists \bar{y}_j. \text{cf}(\mathbb{C}_j)$. The shape analysis sentence (14) is called $\text{SAS}_\Phi(\Pi)$. For the conclusion of $\text{SAS}_\Phi(\Pi)$, i.e. the conclusion of the implication, we write $\text{SASConc}_\Phi(\Pi)$.

6.3 Enrich-by-Need Compares Protocols by Hypotheses

Fortunately, the shape analysis sentences $\text{SAS}_\Phi(\Pi)$ are always in $H(\Phi)$, and the induced partial order $\leq^{H(\Phi)}$ has a nice property relative to $\text{SAS}_\Phi(\Pi)$.

Theorem 2 Π achieves Γ iff $\text{SASConc}_{\text{hyp}(\Gamma)}(\Pi) \Rightarrow \text{conc}(\Gamma)$.

Proof For the forward implication, we rely on the fact that $\text{SASConc}_{\text{hyp}(\Gamma)}(\Pi)$ is the strongest conclusion allowed from $\text{hyp}(\Gamma)$ in Π . That is, whenever $\text{hyp}(\Gamma) \Rightarrow \chi$ then $\text{SASConc}_{\text{hyp}(\Gamma)}(\Pi) \Rightarrow \chi$. So let $\chi = \text{conc}(\Gamma)$.

For the reverse implication, the assumption means that $\Gamma \leq \text{SAS}_{\text{hyp}(\Gamma)}(\Pi)$ in the strength ordering. Also, Π achieves $\text{SAS}_{\text{hyp}(\Gamma)}(\Pi)$ by definition, so by Lemma 1, Π achieves Γ . \square

This theorem says that we can completely characterize $\text{Ach}(\Pi) \cap H(\Phi)$ because it has a single maximum element, namely $\text{SAS}_\Phi(\Pi)$. Thus, we get the following corollary.

Corollary 1 $\Pi_1 \leq^{H(\Phi)} \Pi_2$ iff Π_2 satisfies $\text{SAS}_\Phi(\Pi_1)$.

This means we can use CPSA to compare protocols according to $\leq^{H(\Phi)}$. We simply run the tool once to generate $\text{SAS}_\Phi(\Pi_1)$ and then check whether Π_2 satisfies $\text{SAS}_\Phi(\Pi_1)$ using any semi-decision procedure for finding counterexamples.

We can also define larger sets by considering a set P of hypotheses and taking the union $G = \bigcup_{\Phi \in P} H(\Phi)$. When P is finite we can run the tool once for each member of P . We thus verify that $\Pi_1 \leq^G \Pi_2$, by checking that Π_2 satisfies $\text{SAS}_\Phi(\Pi_1)$ for each $\Phi \in P$.

We can again use the example of the variants of PKINIT to demonstrate the induced ordering. We can let Γ be the desired security goal and run CPSA on both fixes PKINIT₁ and PKINIT₂ starting from inputs representing $\text{hyp}(\Gamma)$. The result is that

$$\text{SASConc}_{\text{hyp}(\Gamma)}(\text{PKINIT}_1) = \text{SASConc}_{\text{hyp}(\Gamma)}(\text{PKINIT}_2).$$

Thus we find that

$$\text{PKINIT} \leq^{H(\Phi)} \text{PKINIT}_1 \bowtie^{H(\Phi)} \text{PKINIT}_2.$$

This is the same ordering as for $\leq^{\{\Gamma\}}$. In this case, increasing the size of the set of goals we consider does not help us further distinguish the protocols under consideration. It does, however, tell us that the two fixes are equally good for a larger class of goals. This assurance would be useful for standards committees. It would allow them to know that both fixes will behave the same under adjustments to the strength of the conclusion of Γ .

6.4 Comparing Protocols by Conclusions

Using the set $H(\Phi)$ uses the idea that fixing a hypothesis, we can compare goals based on the strength of their conclusions. We can also reverse that idea by fixing a conclusion and comparing goals based on the strength of their hypotheses. The intuition is that for two protocols that both achieve a given conclusion, we might prefer the one that requires fewer assumptions. For example, in order for an initiator to authenticate a responder, one protocol may require that the private keys of *both* parties be kept secret, while another protocol may only require that the responder's private key be kept secret. This motivates the use of the following set of goals that share a common conclusion.

$$C(\chi) = \{\Gamma \mid \text{conc}(\Gamma) = \chi\}$$

Thus, for $\Gamma, \Gamma' \in C(\chi)$, $\Gamma \leq \Gamma'$ if and only if $\text{hyp}(\Gamma) \Rightarrow \text{hyp}(\Gamma')$. In order to determine an ordering $\Pi_1 \leq^{C(\chi)} \Pi_2$, we are again faced with the task of trying to compute $\text{Ach}(\Pi_i) \cap C(\chi)$. Unfortunately, there is no simple analog to Theorem 2. $\text{Ach}(\Pi) \cap C(\chi)$ may not have a single maximal element.

Discovering maximal elements of $\text{Ach}(\Pi) \cap H(\Phi)$ relies on a sort of deductive inference in which conclusions are inferred from the hypothesis Φ . The enrich-by-need analysis method naturally supports this type of “forward” inference systematically determining the strongest conclusions allowed from the hypothesis.

Discovering maximal elements of $\text{Ach}(\Pi) \cap C(\chi)$ is a type of *abductive reasoning* in which hypotheses are inferred from the conclusion χ . The enrich-by-need method is not as well suited to support this type of “backward” inference. A better method would be one that systematically determines weaker hypotheses that are still sufficient to imply χ . This would be a sort of “impoverish-as-possible” analysis method. We are unaware of any tools that are specifically designed to support this type of reasoning. Such a tool might enable an analog to Theorem 2 and thus provide a simple way to verify whether $\Pi_1 \leq^{C(\chi)} \Pi_2$. This suggests a potential area for future research.

In the absence of an impoverish-as-possible analysis tool, we believe that $\leq^{C(\chi)}$ is still a useful notion that can help clarify relationships among protocol variants. For one thing, if one finds a sufficiently strong assumption Φ to ensure χ , one can run CPSA repeatedly, omitting conjuncts of Φ until χ is no longer satisfied. Tool support apart, $\leq^{C(\chi)}$ also serves to organize the concepts in a way that enables clearer discussions among standards committee members regarding relative trade-offs. Also, if we are able to use other means to come up with a goal $\Gamma \in C(\chi)$ that Π_1 achieves but

Π_2 does not, then we can use Theorem 1 to demonstrate $\Pi_1 \not\leq^{C(\chi)} \Pi_2$.

7 Conclusion

In this paper we have presented a family of security metrics \leq^G for cryptographic protocols, parameterized by sets of goals G expressed in a logical language. The natural partial ordering on formulas induced by logical implication yields a partial order on protocols that reflects which attacks are possible for a Dolev-Yao style adversary to achieve. This ensures that protocol Π_1 is at least as strong as Π_2 exactly when Π_1 achieves any security goal achieved by Π_2 . We showed how these security metrics can capture and expand well-studied security hierarchies from the literature [26, 14].

The language is designed to be tool-independent, so that any of the wide variety of protocol analysis tools available might be used to verify that a protocol meets a given goal. Although we rely on a well-understood semantics of our language in the strand space formalism, there should be no fundamental barriers to using other formalisms as a semantic base. We believe this is a step toward enabling a diverse set of analysis tools to “speak the same language,” allowing analysts to use different tools based on their relative strengths. This vision is mostly limited by the types of properties we have chosen to consider and the adversary model we use to verify them. We currently only consider trace properties. Thus, our language is not well-suited for indistinguishability properties (useful for privacy goals, among others) or probabilistic properties relying on the more complex details of a computational adversary model. Expanding our ideas to accommodate such goals (and the tools that verify them) is an area of future work.

We also identified the strengths of a particular style of protocol analysis called enrich-by-need. This analysis style is embodied by such tools as our own CPSA as well as Scyther. With such tools we can identify the unique strongest goal achieved by a protocol Π in a particular class of goals denoted $H(\Phi)$ that all share a common hypothesis Φ . We suspect other styles of protocol analysis might be similarly characterized. An interesting area for future investigation would be to develop an “impoverish-as-possible” style of analysis which would identify maximal elements achieved by Π in the set $C(\chi)$ of goals that share a conclusion χ .

We believe this logical formulation of security goals supported by formal verification can help the process of standardizing cryptographic protocols. Standards bodies could adopt the practice of requiring submissions to include explicit claims of goals they achieve—written in

a formal goal language such as ours—together with evidence that those goals have been formally verified with an analysis tool. Our motivating example of PKINIT suggests that the process of explicitly including formal security claims (even without evidence) would go a long way toward improving the quality of published standards as suggested by Basin et al. [5]. It would force the standards bodies to specifically delineate what properties a protocol is designed to achieve, for instance formulas (4) and (7) in the case of PKINIT. The result would be standards that contain concrete artifacts that could be independently verified by others, thereby increasing the public’s confidence in the claims.

Our logical goal language could be useful even in the absence of a formal process for its use. The precision of the language makes it easier to understand the security consequences of subtle changes to a protocol’s design. This can help guide committee members by providing explicit statements and evidence for the positive and negative consequences of design changes, thereby making it easier to make a case for or against a particular design choice. We expect that, used in this way, our goal language could make the standardization process more transparent, with results that are more robust to future attacks.

References

1. Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages (POPL ’01)*, pages 104–115, January 2001.
2. Omar Almousa, Sebastian A. Mödersheim, Paolo Modesti, and Luca Viganò. Typing and compositionality for security protocols: A generalization to the geometric fragment. In *ESORICS*, LNCS. Springer, September 2015.
3. David A. Basin, Cas Cremers, and Simon Meier. Provably repairing the ISO/IEC 9798 standard for entity authentication. *Journal of Computer Security*, 21(6):817–846, 2013.
4. David A. Basin and Cas J. F. Cremers. Modeling and analyzing security in the presence of compromising adversaries. In *ESORICS*, pages 340–356, 2010.
5. David A. Basin, Cas J. F. Cremers, Kunihiko Miyazaki, Sasa Radomirovic, and Dai Watanabe. Improving the security of cryptographic protocol standards. *IEEE Security & Privacy*, 13(3):24–31, 2015.
6. Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *Crypto*, pages 232–249, 1993.
7. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security and Privacy*, 2014.
8. Bruno Blanchet. An efficient protocol verifier based on Prolog rules. In *14th Computer Security Foundations Workshop*, pages 82–96. IEEE CS Press, June 2001.
9. R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *Eurocrypt*, LNCS, pages 453–474. Springer, 2001.

10. Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In *Eurocrypt*, LNCS, pages 337–351. Springer Verlag, 2002.
11. Iliano Cervesato, Aaron D. Jaggar, Andre Scedrov, Joe-Kai Tsay, and Christopher Walstad. Breaking and fixing public-key Kerberos. *Inf. Comput.*, 206(2-4):402–424, 2008.
12. The MITRE Corporation. The common vulnerabilities and exposures (CVE) initiative. <http://cve.mitre.org>.
13. The MITRE Corporation. The common weakness enumeration (CWE). <http://cwe.mitre.org>.
14. Cas Cremers and Sjouke Mauw. *Operational Semantics and Verification of Security Protocols*. Springer, 2012.
15. T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.
16. Daniel J. Dougherty and Joshua D. Guttman. Decidability for lightweight Diffie-Hellman protocols. In *IEEE Symposium on Computer Security Foundations*, 2014.
17. Nancy Durgin, Patrick Lincoln, John Mitchell, and Andre Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004. Initial version appeared in *Workshop on Formal Methods and Security Protocols*, 1999.
18. Santiago Escobar, Catherine Meadows, and José Meseguer. Maude-NPA: Cryptographic protocol analysis modulo equational properties. *Foundations of Security Analysis and Design V*, pages 1–50, 2009.
19. International Organization for Standardization. ISO/IEC 29128: Information technology—security techniques—verification of cryptographic protocols, 2011.
20. Joshua D. Guttman. Shapes: Surveying crypto protocol runs. In Veronique Cortier and Steve Kremer, editors, *Formal Models and Techniques for Analyzing Security Protocols*, Cryptology and Information Security Series. IOS Press, 2011.
21. Joshua D. Guttman. State and progress in strand spaces: Proving fair exchange. *Journal of Automated Reasoning*, 48(2):159–195, 2012.
22. Joshua D. Guttman. Establishing and preserving protocol security goals. *Journal of Computer Security*, 22(2):201–267, 2014.
23. Joshua D Guttman, Moses D Liskov, and Paul D Rowe. Security goals and evolving standards. In *Security Standardisation Research*, pages 93–110. Springer, 2014.
24. ISO/IEC IS 9798-2, "Entity authentication mechanisms – Part 2: Entity authentication using symmetric encipherment algorithms", 1993.
25. Changwei Liu, Anoop Singhal, and Duminda Wijesekera. A model towards using evidence from security events for network attack analysis. In *WOSIS 2014 - Proceedings of the 11th International Workshop on Security in Information Systems, Lisbon, Portugal, 27 April, 2014*, pages 83–95, 2014.
26. Gavin Lowe. A hierarchy of authentication specification. In *CSFW*, pages 31–44, 1997.
27. R. D. Luce and P. Suppes. Measurement, theory of. *Encyclopedia Britannica*, 15th Edition(11):739–745, 1974.
28. Robert A. Martin. Making security measurable and manageable. In *MILCOM 2008*, November 2008.
29. C. Meadows. The NRL protocol analyzer: An overview. *The Journal of Logic Programming*, 26(2):113–131, 1996.
30. C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). RFC 4120 (Proposed Standard), July 2005. Updated by RFCs 4537, 5021, 5896, 6111, 6112, 6113, 6649, 6806.
31. John D. Ramsdell, Daniel J. Dougherty, Joshua D. Guttman, and Paul D. Rowe. A hybrid analysis for security protocols with state. In *Integrated Formal Methods*, pages 272–287, 2014.
32. John D. Ramsdell and Joshua D. Guttman. CPSA: A cryptographic protocol shapes analyzer, 2009. <http://hackage.haskell.org/package/cpsa>.
33. E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), February 2010.
34. A. W. Roscoe. Intensional specifications of security protocols. In *IEEE Computer Security Foundations Workshop*, pages 28–38, 1996.
35. Dawn Xiaodong Song. Athena: A new efficient automated checker for security protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*. IEEE CS Press, June 1999.
36. Kun Sun, Sushil Jajodia, Jason Li, Yi Cheng, Wei Tang, and Anoop Singhal. Automatic security analysis using security metrics. In *MILCOM 2011*, November 2011.
37. F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2/3):191–230, 1999.
38. Thomas Y. C. Woo and Simon S. Lam. Verifying authentication protocols: Methodology and example. In *Proc. Int. Conference on Network Protocols*, October 1993.
39. L. Zhu and B. Tung. Public Key Cryptography for Initial Authentication in Kerberos (PKINIT). RFC 4556 (Proposed Standard), June 2006. Updated by RFC 6112.

Localizing Security for Distributed Firewalls

Pedro Adão¹, Riccardo Focardi²,
Joshua D. Guttman³, and Flaminia L. Luccio⁴

¹ Instituto Superior Técnico, Universidade de Lisboa and
SQIG, Instituto de Telecomunicações

² University Ca' Foscari, Venice and Cryptosense

³ The MITRE Corporation and Worcester Polytechnic Institute

⁴ University Ca' Foscari, Venice

Abstract. In complex networks, filters may be applied at different nodes to control how packets flow. In this paper, we study how to locate filtering functionality within a network. We show how to enforce a set of security goals while allowing maximal service subject to the security constraints. Our contributions include a way to specify security goals for how packets traverse the network and an algorithm to distribute filtering functionality to different nodes in the network to enforce a given set of security goals.

1 Introduction

Organizations have big and complicated networks. A university may have a network partitioned into dozens of subnets, separated either physically or as VLANs. Although many of those subnets are very similar, for instance in requiring similar protection, others are quite distinct, for instance those that contain the university's human resources servers. These require far tighter protection. As another example, consider a corporation: Some subnets contain public-facing machines such as web servers or email servers; others support an engineering department or a sales department; and yet others contain the process-control systems that keep a factory operating. Thus, they should be governed by entirely different policies for what network flows can reach them, and from where.

Indeed, a network is a graph, in which the packets flow over the edges, and the nodes may represent routers, end systems, and so forth. The security goals we would like to enforce reflect this graph structure. They are essentially about trajectories, i.e. about where packets travel to get where they are going. For instance, a packet that reaches the process control system in the factory should not have originated in the public internet. After all, some adversary may use it to insert a destructive command, regardless of how benign its source address header field looks when it arrives. Similarly, a packet that originates in the human resources department should not traverse the public internet en route to the sales department. It could be inspected while there, compromising information about salaries within the company. A security goal may also restrict which packets may take a particular trajectory, for instance only packet addresses to port 80 or 443 on a web server.

Our contribution. In this summary of ongoing work, we describe theories and tools that are under development to protect complex networks by enforcing security policies that control the trajectories of packets through them. Our approach is motivated by some of our own previous work. We have previously studied trajectory-based security goals [3], developing techniques to determine whether existing configurations enforce them correctly. We formalize the network graphs and their possible executions in our frame model [4].

We will here describe the outlines of an approach that starts with a network topology together with a set of security goals to enforce. These security goals constrain which packets may follow a trajectory. The analysis uses the topology to determine what enforcement should be applied at which locations in the network. Our methods are designed to apply also in the case of network operation that transform packets as they pass; we have particularly focused on network address translation (NAT).

In future work we will connect the resulting enforcement strategy to our declarative, order-independent language Mignis which can be compiled to generate concrete implementations under Netfilter [1].

2 Model and Security Goals

2.1 System model

We work within our frame model [4]. Suppose given three domains $\mathcal{LO}, \mathcal{CH}, \mathcal{D}$, to which we will refer as the *locations*, *channels*, and *data*, resp. Each channel will be a unidirectional conduit between the locations that are its endpoints. In a networking context, where flows are frequently bidirectional, these channels can be grouped into pairs.

An *event* e occurs on a channel $\text{chan}(e) \in \mathcal{CH}$ and carries some data $\text{msg}(e) \in \mathcal{D}$. Each channel $c \in \mathcal{CH}$ may be connected to two locations $\text{sender}(c)$ and $\text{rcpt}(c) \in \mathcal{LO}$.

A *frame* is a directed graph where the nodes are locations $\ell \in \mathcal{LO}$, and the arcs are channels $c \in \mathcal{CH}$ connecting $\text{sender}(c)$ to $\text{rcpt}(c)$; moreover, each location is equipped with a labeled transition system. In particular $\ell \in \mathcal{LO}$ has a transition $s \xrightarrow{\ell} s'$ only if $\ell = \text{sender}(c)$ or $\ell = \text{rcpt}(c)$. Thus, every location has a set of traces involving transmissions and receptions on the channels connected to it.

An execution $\mathcal{A} = \mathcal{E}, \preceq$ of a frame is a well-founded partially ordered set of events such that, for every $\ell \in \mathcal{LO}$, the set of events occurring on channels connected to the single location ℓ are in fact linearly ordered by \preceq , and form a possible trace of ℓ . That is, the events involving that location are a possible trace, ordered by the sequence in which they occur.

In [4] we argue that frames and their executions allow effective reasoning about information flow and limited disclosure in distributed systems.

There are many ways to represent networks by frames. We will generally assume that the locations represent routers, end hosts, or network regions. We will

generally assume a pair of arcs, and (in the example) write a single undirected edge. Usually the state of a router or network region includes a set of received but not yet forwarded packets. A state transition in a router may add a new packet; remove and discard a packet (filtering); remove and transmit a packet on a particular edge; or remove, transform, and transmit a packet as in a NAT translation. For simplicity in our example, we assume that a router does not change its filtering rules. Network regions transmit packets but do not filter or transform them.

A *trajectory* in an execution \mathcal{A} is a \preceq -increasing sequence of events of \mathcal{A} that track a single packet as it is transferred from location to location, and possibly transformed by NAT rules.

By a *property ϕ of packets*, we mean a set of packets. Generally, these are characterized by one or more headers of the packets, i.e. as the set of all packets that have specified values for these headers, or unions of such sets.

2.2 Security Goals

We focus on what we will call *three-region policy statements* as security goals. We refer to them as *region control statements*. These take the following form, in which the region variables B, E, R each refer to a network location, and ψ_B, ψ_E , and ϕ refer to sets of packets. These predicates refer to the header fields of the packet at that step in the trajectory, which may vary from its header fields at other steps, in cases such as NAT:

Region control $\psi_B @ B \rightarrow \phi @ R \rightarrow \psi_E @ E$: For every trajectory τ ,
if τ starts at location B with a packet that satisfies ψ_B ,
and τ ends at location E with a packet that satisfies ψ_E ,
then if location R is traversed in τ , the packet satisfies ϕ while at R .

In these region control statements, the sets ψ_B, ψ_E restrict the applicability of the security goal: They constrain a trajectory only if the packet satisfies ψ_B, ψ_E as it exists at the beginning and end respectively. By contrast, ϕ is imposing a requirement, since the network must ensure it is satisfied when the trajectory reaches R .

We can express many useful properties by suitable choices of ϕ . For instance, we may want to ensure that a packet passing from B to E undergoes network address translation properly, so that its source address at the time it traverses R is a *routable address* rather than a private address. We may want to assure that packets from public regions B to a protected corporate region E have been *properly filtered* by the time they reach the corporate entry network R ; thus, they should be `tcp` packets whose destinations are the publicly accessible web and email servers, and whose destination ports are the corresponding well-known ports. These provide examples of region control statements.

We will always assume that $B \neq E$, but there are many useful cases in which the intermediate region R equals one of the endpoints, i.e. $B = R$ or $R = E$. We refer to these as *two-region* statements, since they just restrict the packets

that can travel from B to E . When $R = E$, the statement says that whenever a packet travels from B to R , it must satisfy ϕ . Generally speaking, when the purpose of the statement is to protect R from potentially harmful packets from B , this form of the statement is useful; the property ϕ specifies which packets are safe. The two-region formulas may also be used with $R = B$ to protect B against disclosure of certain packets to E . In this case, the property ϕ specifies which packets are non-sensitive.

Most typical firewall rules are formalized in our framework as two-region rules.

Also of interest are *traversal control statements* $\psi_B@B \rightarrow \phi@R \rightarrow \psi_E@E$. The traversal control statement says that every trajectory from B to E must traverse R and packets must satisfy ϕ while at R ; its applicability is restricted to packets that satisfy ψ_B and ψ_E at the beginning and end, resp.

Traversal control $\psi_B@B \rightarrow \phi@R \rightarrow \psi_E@E$: For every trajectory τ ,
if τ starts at location B with a packet that satisfies ψ_B ,
and τ ends at location E with a packet that satisfies ψ_E ,
then location R is traversed in τ and packets satisfy ϕ while at R .

For instance, consider a corporate network that has packet inspection in a particular region R . Then we may want to ensure that packets from public sources B to internal destinations E traverse R . The reverse is also important in most cases, i.e. that packets from internal sources to public destinations should traverse R .

Given a particular network graph, one strategy to enforce a traversal control statement is using region control statements. We may select a suitable cut set C of nodes between B and E where $R \in C$. We can then enforce a traversal control statement by stipulating the region control statements that for trajectories from B to E : if the packets traverse R they satisfy ϕ ; if the packets traverse any member of $C \setminus \{R\}$, then they satisfy the always-false header property **false**. That is, we have the following family of statements:

$$\begin{aligned} \psi_B@B &\rightarrow \phi@R \rightarrow \psi_E@E \\ \psi_B@B &\rightarrow \mathbf{false}@R' \rightarrow \psi_E@E \quad \forall R' \in C \setminus \{R\} \end{aligned}$$

Given this, we will focus our attention on region control statements $\psi_B@B \rightarrow \phi@R \rightarrow \psi_E@E$.

A trajectory violates a region control statement if it has the correct beginning and end points, but violates the property ϕ while at R .

Functionality Goals. Unlike security goals, which are mandatory, functionality may be a matter of degree. We choose to measure functionality by the set of packets that have a *successful trajectory*. A successful trajectory is one in which a packet travels from a non-spoofing producer to a consumer actually located at the destination address of the packet. We focus on successful trajectories because we regard spoofing originators as intrinsically hostile, which is also the case for promiscuous hosts that consume packets not addressed to them.

We regard one system as *at least as successful functionally* as another system over the same network graph iff, for every successful trajectory permitted by the latter, the same trajectory is permitted by the former.

Given an underlying network topology, formalized as a graph, and a set of security goals, the acceptable systems are those that allow no counterexamples to the security goals. Among those, one would like to construct a frame (specifying the filtering behavior) that is maximal in the ordering of successful functionality.

3 Localizing filtering to enforce goals

Suppose that we are given a set of goal formulas, each a region control statement $\psi_B @ B \rightarrow \phi_0 @ R \rightarrow \psi_E @ E$. We assume a bit of bookkeeping for the forms of each statement. Namely, we assume that each statement concerns either only successful trajectories or else only unsuccessful trajectories. The goal concerns only successful trajectories if $\psi_B \Rightarrow \mathbf{sa}(p) \in IP(B)$ and $\psi_E \Rightarrow \mathbf{da}(p) \in IP(E)$, meaning that the source address of any relevant packet at the start is one of the IP addresses of its actual starting point B , and its destination address at the end is one of the IP addresses of its actual endpoint E . That is, it is not created with a spoofed source address, and it is not consumed by a promiscuous interface to which it is not addressed. By replicating rules, we can rewrite them in a form such that any one rule applies only to non-spoofed, non-promiscuous packets, or alternatively only to packets that are either spoofed or promiscuously delivered. We call these *success rules* and *promiscuity rules* resp.

We first assume that the network involves no NAT rules.

We call this process *localizing* the rules, because we determine which locations to use to enforce those rules.

Localizing success rules without NATs. When the network uses no NAT rules, packets remain the same throughout their trajectories. Thus, in any region R , all of the packets that may traverse R as part of a successful trajectory from B_1 to E_1 will have source address in B_1 and destination address in E_1 . They can never be confused with any packets that are in a successful trajectory from B_2 to E_2 when $B_1 \neq B_2$ or $E_1 \neq E_2$. This observation allows us to compute which packets are useful to keep (for success trajectories) on R separately for each pair of endpoints B, E ; the packets for any other pair are distinguished from them by addresses. Thus, we will do separate computations and then take unions later.

The key intuition is that, for endpoints B, E , we would like to keep those packets at R which:

- are permitted by all the B, E goals to be at R ;
- have a path from B to R touching only permissible regions; and
- have a path from R to E touching only permissible regions.

This is the *keep* set for R , given B, E . We can compute *keep* sets using a matrix computation in the ring of sets of packets, where the ring addition is set union and the ring multiplication is set intersection. This computation is tractable

when the sets are represented via Binary Decision Diagrams, as we found previously [3].

The computation reaches a fixed point because a non-simple path never allows more flow than the simple subpath it contains. Each node may decrease the set of surviving packets by filtering, but cannot increase it.

We must consider all B, E pairs, where a pair that lacks goals is understood as permitting any packets at intermediate nodes. When we have completed the pairs, we have computed all of the packets that should be *kept* in each region R , because those packets have a route traversing R from their unspoofed source address to their intended destination. We call this value $\text{KEEP}(R)$. An appropriate filtering rule for an interface $R' \rightarrow R$ to R from an adjacent R' may discard any packets not in $\text{KEEP}(R)$. For instance, assuming that the packets in R' will be either generated there or else in its $\text{KEEP}(\cdot)$ set, we may filter $(\text{gen}(R') \cup \text{KEEP}(R')) \setminus \text{KEEP}(R)$ on the interface $R' \rightarrow R$.

This computation is optimal in functionality, because it allows all success trajectories that are compatible with the chosen security goals.

Localizing success rules with NATs. Curiously, the computation for the case where the network has NAT rules is very similar, but is performed in a different ring. Namely, we are no longer interested in a set of packets, but in a relation between packets in their state at a previous location and packets in their state at a later location. Thus, the matrices A will contain, in entry $A_{i,j}$, a relation between packets that may have occurred at location i and their resulting state when reaching location j . The addition in this ring is union. The multiplication is relational product. That is, the “product” of the binary relations $S(x, y)$ and $S'(y, z)$ is the binary relation $\exists y. S(x, y) \wedge S'(y, z)$. Thus, we are no longer working in a commutative ring. However, we have simply lifted the set-based computation to a relation-based computation.

Binary Decision Diagrams may still be used, although the relative product computation for $\exists y. S(x, y) \wedge S'(y, z)$ requires taking cases on the boolean variables that contributing to the projected variable y . We do not yet have an estimate of the cost of this computation.

There is an assumption to be made here, to ensure the analogue to the independence of the packets for different beginnings and endpoints. When a number of beginnings are behind the same source NAT, the security goals for regions R beyond the NAT should treat them the same way. Nothing else can be enforced, since they will be indistinguishable beyond the NAT. Destinations behind the same destination NAT must be treated uniformly for a similar reason.

A second contrast with the no-NAT case concerns the simple path assumption. NATs can be arranged so that a non-simple path would produce new packets, although this is contrary to the main purposes for which NATs are used. Thus, we terminate when the computation reaches its fixed point. Some modifications would not reach a fixed point in limited time. For instance, suppose a router applied a bizarre form of NAT in which it increments the source address of the packet. If the network might cause that packet to traverse the router repeatedly, then its source address will be incremented repeatedly. However, for reasonable

transformations, although the fixed point is not guaranteed to happen quickly, it is extremely likely in fact to do so.

Simple paths. If one desires, one can adapt the above computations to be based only on simple paths. A classical technique [7], adapted to our context, is to tag sets of packets with the set of locations that have figured in the relevant paths. When combining paths say $B \rightarrow^+ R$ and $R \rightarrow^+ E$, one checks that R is the only location they have both visited. This further complicates the data structure and requires the computation to handle more cases separately, namely all those where paths traverse different set of locations. However, this tagging scheme actually lifts a ring to a more complex ring, thus allowing the same structure for the core computations.

Localizing promiscuity rules. Once we have computed the filtering for the success rules, the promiscuity rules may already be satisfied. This will happen whenever the intermediate locations needed for success trajectories are disjoint from the suspicious locations that the promiscuity rules are protecting against.

Otherwise, there is genuine conflict between functionality for permitted success trajectories and security concerns for messages that might be spoofed in locations they traverse. There is not always a canonical, best-functionality solution to this problem. However, we can always enforce a set of promiscuity rules by using another matrix computation. We find the set of promiscuous trajectories that are permitted by the existing, success-based filtering rules. For all packets that can traverse a trajectory, we then update the filters on a link of the trajectory to discard these packets. An attractive heuristic is that for anti-spoofing goals, we should discard the packets as early as possible: We do not want them to get anywhere. However, for anti-promiscuous delivery goals, we should discard them as late as possible. These packets, if safely routed, will be useful.

Simplifying the network topology. The computations we are describing are tractable, because in fact we can simplify the topology of networks. In particular, given a large network with a set of filtering locations identified, we can shrink the network by identifying devices that are in similar positions relative to the filtering locations. In particular, two devices should be mapped to the same region when they have simple paths to exactly the same set of filtering locations, where these paths traverse no intermediate filtering locations.

Using this idea, complicated networks actually furnish small graphs for our algorithms to work with.

4 Case study

We consider the case study depicted in Figure 1 composed of three subnetworks: **Sensitive**, **Trusted** and **Untrusted**. **Sensitive** subnetwork contains important servers and data and is connected to the **Internet** through the firewall router **fw1** and then gateway **gw1**; **Trusted** subnetwork is composed of trusted

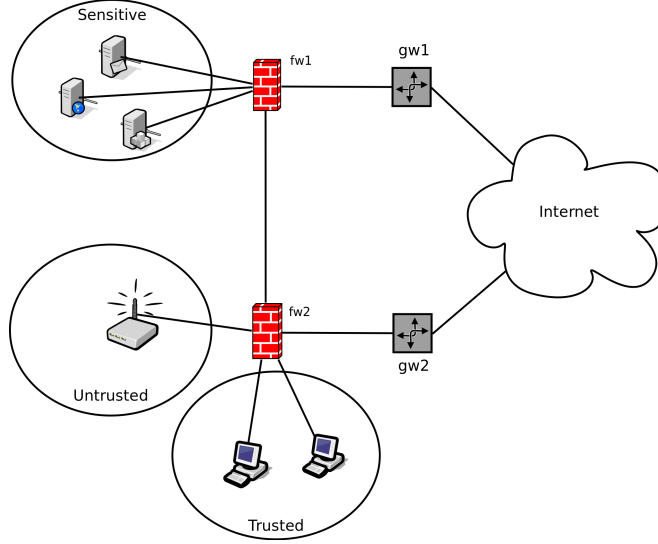


Fig. 1. A simple network with two firewall routers **fw1** and **fw2**.

hosts that, for example, can access services hosted in the **Sensitive** subnetwork; **Untrusted** is a wifi subnetwork providing a controlled access to the **Internet** but not to services hosted in **Sensitive**. Both **Trusted** and **Untrusted** are connected to the firewall router **fw2** which in turn is connected to the other firewall router **fw1**, and to the **Internet** through gateway **gw2**.

We now define security goals for the example network.

Success rules. The following rules apply only to non-spoofed, non-promiscuous packets. This can be easily enforced by assuming ψ_B and ψ_E respectively check if $\text{sa}(p) \in IP(B)$ and $\text{da}(p) \in IP(E)$. For the sake of readability we omit to write these checks in the rules and we leave them implicit.

Firewalls usually keep track of established connections so that packets belonging to the same connections are not filtered. This is particularly useful to enable bidirectional communication without necessarily opening the firewall bidirectionally to new connection: it is enough to open the firewall in one direction and let *established* packets come back. In the following we write **est** to note that a packet is established. While specifying rules, we proceed pair by pair so to define rules and their established counterpart (when needed) at the same time.

Hosts in the **Sensitive** and **Trusted** subnetworks should never connect to **Untrusted** and vice-versa. This is naturally expressed through two-region statements in which R corresponds to B or E (cf. Section 2.2):

```

Sensitive → false@Sensitive → Untrusted
Untrusted → false@Sensitive → Sensitive
Trusted → false@Trusted → Untrusted
Untrusted → false@Trusted → Trusted

```

Hosts in the **Sensitive** subnetwork should never connect to **Trusted**, while hosts from **Trusted** network can access **Sensitive** via ssh through **fw1** without passing through the **Internet** as this would unnecessarily expose network connections to attacks. Notice that we filter packets from **Sensitive** to **Trusted** only if they do not belong to established ssh connections. This is achieved by adding a precondition on the start point in the second rule below:

$$\begin{array}{llll}
& \text{Trusted} \rightarrow \text{dport} = 22 @ \text{Sensitive} \rightarrow \text{Sensitive} \\
\neg(\text{sport} = 22 \wedge \text{est}) @ \text{Sensitive} \rightarrow & \text{false} @ \text{Sensitive} & \rightarrow \text{Trusted} \\
& \text{Trusted} \rightarrow \text{false} @ \text{gw1, gw2} & \rightarrow \text{Sensitive} \\
& \text{Sensitive} \rightarrow \text{false} @ \text{gw1, gw2} & \rightarrow \text{Trusted}
\end{array}$$

Sensitive should access the **Internet** only via https (destination port should be 443), while **Internet** hosts should never connect to **Sensitive**:

$$\begin{array}{llll}
& \text{Sensitive} \rightarrow \text{dport} = 443 @ \text{Sensitive} \rightarrow \text{Internet} \\
\neg(\text{sport} = 443 \wedge \text{est}) @ \text{Internet} \rightarrow & \text{false} @ \text{Sensitive} & \rightarrow \text{Sensitive}
\end{array}$$

Trusted has full access to the **Internet** and from the **Internet** we give access to **Trusted** only via ssh (port 22):

$$\neg \text{est} @ \text{Internet} \rightarrow \text{dport} = 22 @ \text{Trusted} \rightarrow \text{Trusted}$$

Untrusted should access the **Internet** exclusively through **gw1** under filter ϕ . This is a form of traversal control that can be compiled into region control rules by taking cut $\{\text{gw1}, \text{gw2}\}$ and forbidding traversal of everything but **gw1**, i.e., **gw2** (cf. Section 2.2). In a real setting, this might be motivated by the fact **fw1** is more powerful than **fw2** being able to handle complex (stateful) protocols covered by ϕ and offering logging capabilities that are useful to check what the untrusted users do. **Internet** should never access **Untrusted**. We let $\bar{\phi}$ denote the counterpart of ϕ holding on established packets (e.g., swapping source and destination ports):

$$\begin{array}{llll}
& \text{Untrusted} \rightarrow \phi @ \text{gw1} & \rightarrow \text{Internet} \\
\neg(\bar{\phi} \wedge \text{est}) @ \text{Internet} \rightarrow & \text{false} @ \text{gw1} & \rightarrow \text{Untrusted} \\
& \text{Untrusted} \rightarrow \text{false} @ \text{gw2} & \rightarrow \text{Internet} \\
& \text{Internet} \rightarrow \text{false} @ \text{gw2} & \rightarrow \text{Untrusted}
\end{array}$$

Promiscuity rules. We assume that subnetworks **Sensitive** and **Trusted** do not spoof or promiscuously deliver packets. Let $N \in \{\text{Sensitive}, \text{Trusted}, \text{Internet}\}$ and $N' \in \{\text{Sensitive}, \text{Trusted}, \text{Untrusted}\}$. The following rules prevent spoofing from **Untrusted** and **Internet**:

$$\begin{array}{ll}
\text{sa} \notin IP(\text{Untrusted}) @ \text{Untrusted} \rightarrow \text{false} @ \text{Untrusted} \rightarrow N \\
\text{sa} \notin IP(\text{Internet}) @ \text{Internet} \rightarrow \text{false} @ \text{Internet} \rightarrow N'
\end{array}$$

while the following ones prevent promiscuous deliver to **Untrusted** and **Internet**:

$$\begin{array}{ll}
N \rightarrow \text{false} @ \text{Untrusted} \rightarrow \text{da} \notin IP(\text{Untrusted}) @ \text{Untrusted} \\
N' \rightarrow \text{false} @ \text{Internet} \rightarrow \text{da} \notin IP(\text{Internet}) @ \text{Internet}
\end{array}$$

Localizing filtering. We show how some of the above goals are localized. We first consider the case of completely forbidden communication from **Sensitive** to **Untrusted**:

$$\text{Sensitive} \rightarrow \text{false@Sensitive} \rightarrow \text{Untrusted}$$

By performing a matrix computation we easily obtain that $\text{KEEP}(\cdot) = \emptyset$ for each node in the network. Indeed, the rule forbids any packet from **Sensitive** to **Untrusted** directly at the source. Since we have

$$\text{gen}(\text{Sensitive}) = \{p \mid \text{sa}(p) \in IP(\text{Sensitive}), \text{da}(p) \in IP(\text{Untrusted})\}$$

then we only need to filter $(\text{gen}(\text{Sensitive}) \cup \text{KEEP}(\text{Sensitive})) \setminus \text{KEEP}(\text{fw1}) = \text{gen}(\text{Sensitive})$ on the interface $\text{Sensitive} \rightarrow \text{fw1}$. All packets from **Sensitive** to **Untrusted** will be dropped as early as possible in **fw1** and no filtering will be done in **fw2**.

We now consider the more interesting situation of traversal control from **Untrusted** to the **Internet** imposing packets to go through **fw1**:

$$\begin{aligned} \text{Untrusted} &\rightarrow \phi@\text{gw1} \rightarrow \text{Internet} \\ \text{Untrusted} &\rightarrow \text{false@gw2} \rightarrow \text{Internet} \end{aligned}$$

In this case we obtain that $\text{KEEP}(\cdot)$ is equal to $\text{gen}(\text{Untrusted})$ for **Untrusted**, **Sensitive**, **fw1**, **fw2**; it is equal to ϕ for **gw1**, **Internet**; it is the emptyset for **gw2**. We thus need to filter $\text{gen}(\text{Untrusted})$ on the interface $\text{fw2} \rightarrow \text{gw2}$ and $\text{gen}(\text{Untrusted}) \setminus \phi$ on the interface $\text{fw1} \rightarrow \text{gw1}$. Intuitively, **fw1** will filter all generated packets that do not belong to the set of permitted packets ϕ while **fw2** will filter any (generated) packet, enforcing traversal control.

In a similar way, we can filter spoofing and promiscuous delivery. For example:

$$\text{sa} \notin IP(\text{Untrusted})@\text{Untrusted} \rightarrow \text{false@Untrusted} \rightarrow \text{Sensitive}$$

will produce a filter on interface $\text{Untrusted} \rightarrow \text{fw2}$ dropping any packet originated in **Untrusted** with a spoofed IP (not in $IP(\text{Untrusted})$) and directed to **Sensitive**. Notice that filtering happens as soon as possible. Dually rule:

$$\text{Sensitive} \rightarrow \text{false@Untrusted} \rightarrow \text{da} \notin IP(\text{Untrusted})@\text{Untrusted}$$

will produce a filter on interface $\text{fw2} \rightarrow \text{Untrusted}$ dropping any packet originated in **Sensitive** and promiscuously delivered to **Untrusted** to an IP not in $IP(\text{Untrusted})$. Notice that in this case filtering happens as late as possible, only in case the packet is (wrongly) routed to **Untrusted**.

5 Conclusion

Our localization strategy does not generate concrete rule-sets for actual devices. However, we plan to generate rules in the declarative, order-independent language of Mignis [1]; the semantically motivated Mignis compiler then generates

concrete rule-sets for Netfilter. This will complete the task of generating semantically correct network configurations from security goal statements.

Our work is distinguished by its focus on clear behavioral security specifications. In this it contrasts with otherwise very strong work, for instance on security using programming language techniques as in NetKAT [2]. Zhang et al. [8] focus more on the possible conflicts among policies at different organizational levels, and less on their consequences given the topology of the network. Much work has been devoted to firewall analysis, e.g. Margrave [6], which again lacks the distributed behavior of the network.

Kurshid et al. [5] demonstrate that it is possible, in a software defined networking context, to check dynamically to ensure that global, behavioral properties are maintained as invariants, for instance reachability for certain sorts of packets. We instead make no claims of real-time, on-line feasibility, but we offer a more systematic way to solve well-defined security problems at design time.

References

1. P. Adao, C. Bozzato, R. Focardi G. Dei Rossi, and F.L. Luccio. Mignis: A semantic based tool for firewall configuration. In *IEEE Computer Security Foundations*, pages 351–365. IEEE CS Press, July 2014.
2. C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*. ACM, 2014.
3. J.D. Guttman and A.L. Herzog. Rigorous automated network security management. *International Journal for Information Security*, 5(1–2):29–48, 2005.
4. J.D. Guttman and P.D. Rowe. A cut principle for information flow. In *IEEE Computer Security Foundations*, pages 107–121. IEEE CS Press, July 2015.
5. Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and PB Godfrey. Veriflow: verifying network-wide invariants in real time. *ACM SIGCOMM Computer Communication Review*, 42(4):467–472, 2012.
6. T. Nelson, C. Barratt, D.J. Dougherty, K. Fisler, and S. Krishnamurthi. The margrave tool for firewall analysis. In *Proceedings of the 24th International Conference on Large Installation System Administration (LISA’10)*, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.
7. F. Rubin. Enumerating all simple paths in a graph. *IEEE Trans. Circuits and Systems*, 25(8):641–642, 1978.
8. B. Zhang, E. Al-Shaer, R. Jagadeesan, J. Riely, and C. Pitcher. Specifications of a high-level conflict-free firewall policy language for multi-domain networks. In *Proc. of ACM Symposium on Access Control Models and Technologies (SACMAT 2007)*. ACM, 2007.

Securing IoT communications: at what cost?^{*}

Chiara Bodei and Letterio Galletta

Dipartimento di Informatica, Università di Pisa
{chiara,galletta}@di.unipi.it

Abstract. IoT systems use wireless links for local communication, where locality depends on the transmission range and include many devices with low computational and battery power such as sensors. In IoT systems, security is a crucial requirement, but difficult to obtain, because standard cryptographic techniques have a cost that is usually unaffordable. We resort to an extended version of the process calculus *LYSA*, called *IoT-LYSA*, to model the patterns of communication of IoT devices. Moreover, we assign rates to each transition to infer quantitative measures on the specified systems. The derived performance evaluation can be exploited to establish the cost of the possible security counter-measures.

1 Introduction

Nowadays, “software is eating the world”, i.e. software is pervading our everyday life and the objects we use such as webTV, coffeemakers, cars, smartphones, ebook readers, and Smart Cities, on a broader scale [3].

The main distinguishing feature of this scenario, called Internet of Things (IoT), is that in principle objects are always connected to the Internet and are equipped with different kinds of sensors, e.g. accelerometers, light sensors, microphone, and so on. These smart devices automatically collect information of various kinds and store them on the cloud or use them to directly operate on the surrounding environment through actuators. For instance, our smart alarm clock can drive our heating system to find a warm bathroom in the morning, while an alarm sensor in our place can directly trigger an emergency call to the closest police station. As a further example consider a storehouse stocking perishable food equipped with sensors to determine the internal temperature and other relevant attributes. The refrigeration system can automatically adapt the temperature according to the information collected by sensors.

The IoT paradigm introduces new pressing security challenges. Although security should be co-designed with the system and not just added

^{*} Work partly supported by project PRA_2016_64 “Through the fog” funded by the University of Pisa.

on as an optional equipment, this is not easy in a setting where the usual trade-off between highly secure and system usability is more critical than ever. For instance, sensors are plagued by several security vulnerabilities: e.g. an attacker can easily intercept sensor communications and manipulate and falsify data. Nevertheless, security costs are high for devices with limited computational and communication capabilities and with limited battery power, and designers must be selective in choosing security mechanisms. Back to the refrigerator system above, to protect against falsification of data by an attacker, it is possible to resort to cryptography and to consistency data checks to prevent and detect anomalies. But is it affordable to secure all the communications? Can we obtain an affordable solution still having a good level of security by protecting only part of communications?

To relieve this kind of problems, IoT designers not only need tools to assess possible risks and to study countermeasures, but also methodologies to estimate their costs. The cost of security can be computed in terms of time overhead, energy consumption, bandwidth, and economic, and so on. All these factors must be carefully evaluated to achieve an acceptable balance between security, cost and usability of the system.

Usually, formal methods provide designers with tools to support the development of systems; also, they support them in proving properties, both qualitative and quantitative. The application of formal methodologies to securing IoT systems is still in its infancy. We would like to contribute by proposing IoT-LySA, an extension of the process calculus LySA [7] (a close relative of the π -[25] and Spi-calculus [1]) to model the patterns of communication of IoT devices, and to infer quantitative measures of the costs the specified systems bear for security mechanisms.

Here, we present preliminary steps, based on [6], towards the development of a formal methodology that supports designers in analysing the cost of security in IoT systems. Our long term goal is to provide a general framework with a mechanisable procedure (with a small amount of manual tuning), where quantitative aspects are symbolically represented by parameters. The instantiation of parameters is delayed until hardware architectures and cryptographic algorithms are fixed. By changing only these parameters designers could compare different implementations of IoT systems and could choose among different alternatives by selecting the better tradeoffs among security and costs. Technically, we extend LySA (Sect. 2) with suitable primitives for describing the activity of sensors and of sensor nodes, and for managing the coordination and communication capabilities of smart objects. The calculus is then given an

enhanced semantics, following [15], where each transition is associated to a rate in the style of [27]. It suffices to have information about the activities performed by the components of a system in isolation, and about some features of the network architecture. Starting from rates, it is possible to mechanically derive Markov chains. The actual cost of security can be carried out using standard techniques and tools [33,30,32]. Note that here quantitative measures can live together with the usual qualitative semantics, where instead these aspects are usually abstracted away. For the sake of simplicity, we do not consider actuators and temporal concerns.

The paper is organised as follows. In Sect. 2, we briefly introduce the process calculus IoT-LySA. In Sect. 3, we present a sample function that assigns rates to transitions, and we show how to obtain the CTMC associated with a given system of nodes and how to extract performance measures from it. Concluding remarks and related work are in Sect. 4.

2 IoT-LySA and its Enhanced Semantics

The original LySA calculus [5,7] is based on the π -[25] and Spi-calculus [1]. The main differences are: (i) the absence of channels, there is only one global communication medium to which all processes have access; (ii) the pattern matching tests associated with received and decrypted values are incorporated into inputs and into decryptions. Below, we assume that the reader is familiar with the basics of process calculi. We extend LySA to model IoT communications by introducing: (i) systems of nodes, consisting of (a representation of the) physical components, i.e. sensors, and of software control processes for specifying the *logic* of the node; (ii) primitives for reading from sensors also with cryptographic protection; (iii) global variables, i.e. whose scope is the whole node, to store data sent by sensors; (iv) a multi-communication modality among nodes (communications are subject to various constraints mainly about proximity); (v) functions to process and aggregate data, in particular the sensor's ones.

Syntax. As shown below, IoT-LySA systems have a two-level structure and consist of a fixed number of uniquely labelled nodes $N \in \mathcal{N}$ that host control processes $P \in \mathcal{P}$, and indexed sensor processes $S_i \in \mathcal{S}$ with $i \in \mathcal{I}_\ell$. Here \mathcal{V} denotes the set of values, while \mathcal{X} and \mathcal{Z} and the local and the global variables, respectively. Finally, \mathcal{L} denotes the set of node labels.

$N ::= \text{systems of nodes}$	
0	nil
$\ell : [P \parallel S]$	single node ($\ell \in \mathcal{L}$)
$N_1 \mid N_2$	parallel composition of nodes

Intuitively, the null inactive system of nodes is denoted by 0 (*nil*); in a single node $\ell : [P \parallel S]$ the label ℓ uniquely identifies the node and represents further characterising information (e.g. its location or other contextual information if needed). Node components are obtained by the parallel composition (through the operator \parallel) of control processes P , and of a fixed number of (less than $\#(\mathcal{I}_\ell)$) sensors S . The syntax of control processes follows.

$P ::=$	<i>control processes</i>	
0		nil
$\langle E_1, \dots, E_k \rangle . P$		intra-node output
$\langle\langle E_1, \dots, E_k \rangle\rangle \triangleright L . P$		multi-output $L \subseteq \mathcal{L}$
$(E_1, \dots, E_j; x_{j+1}, \dots, x_k) . P$		input (with match.)
$P_1 \parallel P_2$		parallel composition of processes
$P_1 + P_2$		summation
$(\nu n)P$		restriction
$A(y_1, \dots, y_n)$		recursion
decrypt E as		decryption (with match.)
$\{E_1, \dots, E_j; x_{j+1}, \dots, x_k\}_{E_0}$ in P		
$(i; z_i) . P$		clear input from sensor i
$(\{i; z_i\}_K) . P$		crypto input from sensor i

The process 0 represents the *inactive* process. The process $\langle E_1, \dots, E_k \rangle . P$ sends the tuple E_1, \dots, E_k to another process in the same node and then continues like P . The process $\langle\langle E_1, \dots, E_k \rangle\rangle \triangleright L . P$ sends the tuple E_1, \dots, E_k to the nodes whose labels are in L and evolves as P . Only nodes that are “compatible” (according, among other attributes, to a proximity-based notion) can communicate. The process $(E_1, \dots, E_j; x_{j+1}, \dots, x_k) . P$ is willing to receive a tuple E'_1, \dots, E'_k with the same arity. The input primitive tests the first j terms of the received message; if they pairwise match with the first j terms of the input tuple, then the variables x_{j+1}, \dots, x_k , occurring in the input tuple are bound to the corresponding terms E_{j+1}, \dots, E_k in the output tuple. The continuation is therefore $P\{E_{j+1}/x_{j+1}, \dots, E_k/x_k\}$, where $\{-/-\}$ denotes, as usual, the standard substitution. Otherwise, the received tuple is not accepted. Note that for simplicity, all the pattern matching values precede all the binding variables: syntactically, a semi-colon separates the two components (see [9,4] for a more flexible choice). To better understand this construct, suppose to have a process P waiting for a message that P knows to include the value v and a value that P still does not know. The input pattern tuple would be: $(v; x'_v)$. If P receives the matching tuple $\langle v, v' \rangle$, the pattern matching

succeeds and the variable x'_v is bound to v' . The operator \parallel describes parallel composition of processes, while $+$ denotes non-deterministic choice. The operator (νa) acts as a static declaration for the name a in the process P the restriction prefixes. Restriction can be used to create new values, e.g. keys. An agent is a static definition of a parameterised process. Each agent identifier A has a unique defining equation of the form $A(y_1, \dots, y_n) = P$, where y_1, \dots, y_n are distinct names occurring free in P . The process **decrypt** E as $E_1, \dots, E_j; x_{j+1}, \dots, x_k\}_{E_0}$ in P receives an encrypted message. Also in this case we use the pattern matching but additionally the message $E = \{E'_1, \dots, E'_k\}_{E'_0}$ is decrypted with the key E_0 . Hence, whenever $E_i = E'_i$ for all $i \in [0, j]$, the receiving process behaves as $P\{E_{j+1}/x_{j+1}, \dots, E_k/x_k\}$.

Sensors have the form:

$S ::= \text{sensor processes}$	
$\tau.S$	internal action
$\langle i, v \rangle . S$	output of the i^{th} sensor
$\langle \{i, v\}_K \rangle . S$	encrypted output of the i^{th} sensor
$S_1 \parallel S_2$	parallel composition of sensors
$A(y_1, \dots, y_n)$	recursion

A sensor can perform an internal action τ or send a (encrypted) value v , gathered from the environment, to its controlling process and continues as S . We do not provide an explicit operation to read data from the environment but it can be easily implemented as an internal action.

Finally, the syntax of term is:

$E ::= \text{terms}$	
v	value ($v \in \mathcal{V}$)
x	variable ($x \in \mathcal{X}$)
z	sensor's variable ($z \in \mathcal{Z}$)
$\{E_1, \dots, E_k\}_{E_0}$	encryption ($k \geq 0$)
$f(E_1, \dots, E_n)$	function application ($n \geq 0$)

A value represents a piece of data, in particular we use them for keys, integers and values read from the environment. As said above, we have two kinds of disjoint variables: x are standard variables, i.e. as used in π -calculus; sensor variables z belong to a node and are globally accessible within it. As usual, we require that variables and names are disjoint. The encryption function $\{E_1, \dots, E_k\}_{E_0}$ returns the result of encrypting values E_i for $i \in [1, k]$ with the key E_0 . The term $f(E_1, \dots, E_n)$ is the application of function f to n arguments; we assume given a set of primitive

aggregation functions, e.g. functions for comparing values or computing some metrics.

Working Example. Consider the scenario presented in the Introduction and illustrated in Fig. 1. We want to set a simple IoT system up in order to keep the temperature under control inside a storehouse with perishable food (a big quadrangular room). We plan to install four sensors: one for each corner of the storehouse. We assume that each sensor S_i periodically senses the temperature and sends it with a wireless communication to a control unit P_c in the same node, which aggregates the read values and checks if the average temperature is within accepted bounds. If this is not the case, the control unit sends an alarm through other nodes and the Cloud. We assume that an attacker can intercept and manipulate data sent by sensors. A possible countermeasure is to exploit the fact that sensors on the same side should sense the same temperature, with a difference that can be at most a given value ϵ . The control unit can indeed easily detect anomalies and discard a piece of data manipulated by an attacker, by comparing it with values coming from the other sensor on the same side. But what if the attacker falsifies the data sent by more than one sensor? A possible solution consists in enabling a part of the sensors (in our example e.g. S_1 and S_3) to use cryptography in order to have at least two reliable data. Nevertheless, before adopting it or evaluating further solutions we would like to estimate the overhead cost. Sensors can be modelled in IoT-LySA as follows.

$$\begin{aligned} S_m &= \langle m, \text{sense}_j = m() \rangle. \tau. S_m & m = 0, 2 \\ S_j &= \langle \{j, \text{sense}_j()\}_{K_j} \rangle. \tau. S_j & j = 1, 3 \end{aligned}$$

The control process P_c of the first node reads from sensors and then aggregates and compares the sensed values, in order to check them and compute their average. The control process Q_c of the second node verifies the result of the comparison and of the average functions and decides if sending an **alarm** or an **ok** message to the third node, together with the average value. The control process R_c of the third node represents an Internet service where the control process waits for the message of the second node and handles it (through the internal action τ). The specifications of the control processes follow.

$$\begin{aligned} P_c &= (0; z_0). \tau. (\{1; z_1\}_{K_1}). \tau. (2; z_2). \tau. (\{3; z_3\}_{K_3}). \tau. \\ &\quad \langle \langle \text{cmp}(z_0, \dots, z_3), \text{avg}(z_0, \dots, z_3) \rangle \rangle \triangleright \{\ell_2\}. \tau. P_c \\ Q_c &= (\text{true}; x_{\text{avg}}). \langle \langle \text{ok}, x_{\text{avg}} \rangle \rangle \triangleright \{\ell_3\}. \tau. Q_c + \\ &\quad (\text{false}; x_{\text{avg}}). \langle \langle \text{alarm}, x_{\text{avg}} \rangle \rangle \triangleright \{\ell_3\}. \tau. Q_c \\ R_c &= (; w_{\text{res}}, w_{\text{avg}}). \tau. R_c \end{aligned}$$

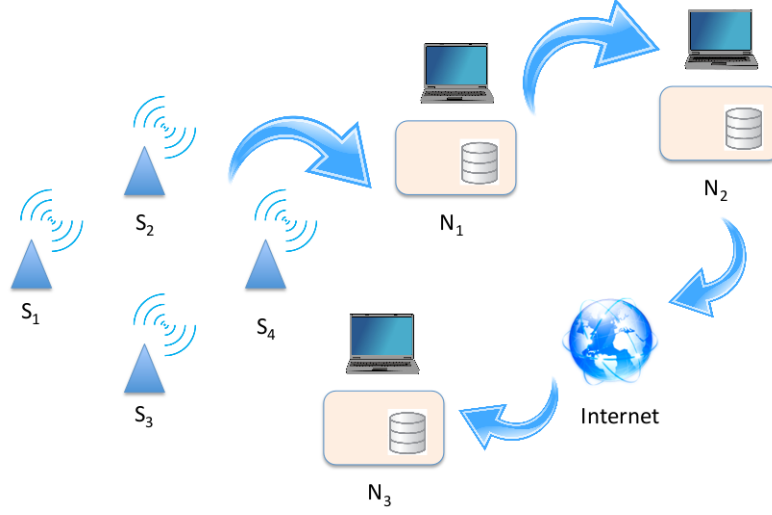


Fig. 1. The organisation of nodes in our refrigerator system.

The aggregation function *cmp* on the collected data perform consistency checks, by comparing data coming from insecure sensors with data coming from sensor endowed with encrypted communication; if the first data are out of bounds, the result is **true** otherwise is **false**. The function *avg* computes the average of its arguments. We suppose that processes and sensors perform some internal activities (denoted by τ -actions). The whole IOT-LYSA node system, which includes the node N_1 (composed by the control unit and sensors) and the nodes N_2 and N_3 , is specified as follows.

$$\begin{aligned}
 N &= N_1 \mid N_2 \mid N_3 & N_2 &= \ell_2 : [Q_c \parallel 0] \\
 N_1 &= \ell_1 : [P_c \parallel (S_0 \parallel S_1 \parallel S_2 \parallel S_3)] & N_3 &= \ell_3 : [R_c \parallel 0]
 \end{aligned}$$

Another solution consists in enabling just one sensor on four to use cryptography. In the new system of nodes \hat{N} , the only difference is the following specification of the control process \hat{P}_c of the first node.

$$\begin{aligned}
 \hat{P}_c &= (0; z_0). \tau. (\{1; z_1\}_{K_1}). \tau. (2; z_2). \tau. (3; z_3). \tau. \\
 &\quad \langle\langle halfcmp(z_0, z_1), avg(z_0, \dots, z_3) \rangle\rangle \triangleright \{\ell_2\}. \tau. \hat{P}_c
 \end{aligned}$$

Note that the required comparison function *halfcmp* is simpler, since it uses only two arguments. We expect that this second solution is less expensive. Our methodology allows us to formally compare the relative costs of the two solutions.

Enhanced Operational Semantics. Here, we give a reduction semantics in the style of the one of LYSA [7]. It is an *enhanced* semantics, because following [14,15], transitions are annotated with labels used to estimate costs (actually, ours is a simplified version of the one in [14,15]).

The underlying idea is that each transition is enriched with an *enhanced label* θ , which records both the actions related to the transition and the possible nodes involved. Actually, there is a label for transitions involving communications and decryptions. More in detail, both point to point communications and multi-communications records the two actions (input and output) that lead to the transition, together with the labels of the corresponding nodes. Decryption actions store the label of the node performing the operation. Note that in the following definition and in the semantic rules, we use the abbreviations *out*, *in*, *dec*, for denoting the communication prefixes, the decryption constructs and, inside them, the possible function calls *f*, of the considered transition. The standard semantics can be obtained by simply removing the transition labels.

Definition 1. (*Transition labels*) Given $\ell_O, \ell_I, \ell_D \in \mathcal{L}$, the set $\Theta \ni \theta$ of enhanced labels is defined as follows.

$$\begin{aligned} \theta ::= & \\ & \langle \ell \{out\}, \ell \{in\} \rangle \quad \text{internal secure communication} \\ & \langle \ell \text{ out}, \ell \text{ in} \rangle \quad \text{internal communication} \\ & \langle \ell_O \text{ out}, \ell_I \text{ in} \rangle \quad \text{intra-nodes communication} \\ & \{ \ell_D \text{ dec} \} \quad \text{decryption of a message} \end{aligned}$$

As usual, our semantics consists of the standard structural congruence \equiv on nodes, processes and sensors and of a set of rules defining the transition relation. Our notion of *structural congruence* \equiv is standard except for the following congruence rule for processes that equates a multi-output with empty set of receivers to the inactive process.

$$\langle \langle E_1, \dots, E_k \rangle \rangle \triangleright \emptyset.0 \equiv P$$

As usual, our *reduction relation* $\xrightarrow{\theta} \subseteq \mathcal{N} \times \mathcal{N}$ is defined as the least relation on closed nodes, processes and sensors that satisfies a set of inference rules. Our rules are quite standard apart from the five rules for communications shown in Tab. 1 and briefly commented below. We assume the standard denotational interpretation for evaluating terms $\llbracket E \rrbracket$.

- the rule (*Sens-Com*) is used for communications between sensors and processes. The used variables are assumed to be *global*. i.e. shared

among the process of the same node. The idea is that sensors contribute to a sort of shared data structure z_1, \dots, z_n . Therefore the substitution is performed on all the control processes in the node. The transition label $\langle \ell \text{ out}, \ell \text{ in} \rangle$ records the fact that an internal communication occurred inside the node ℓ ;

- similarly, the rule (*Crypto-Sens-Com*) is used for protected communications between sensors and processes: the value sensed by the sensor is encrypted before being sent to the process and is received if successfully decrypted. Also in this case the transition label $\langle \ell \{out\}, \ell \{in\} \rangle$ records information about the internal protected communication;
- the rule (*Intra-Com*) is used for communications internal to a node. The communication succeeds, provided that the first j values match with the evaluations of the first j terms in the input. When these comparisons are successful each E_i is bound to each x_i . The transition label $\langle \ell \text{ out}, \ell \text{ in} \rangle$ records the fact that an internal communication occurred inside the node ℓ ;
- the rule (*Point2Point-Com*) is used for point to point communications between nodes. The communication succeeds, provided that (i) the labels of the two nodes are compatible according to the compatibility function *Comp*; and (ii) the first j values match with the evaluations of the first j terms in the input. When these comparisons are successful each E_i is bound to each x_i . The transition label $\langle \ell_1 \text{ out}, \ell_2 \text{ in} \rangle$ records the fact that an inter-node communication occurred between the nodes labelled by ℓ_1 and ℓ_2 ;
- the rule (*Multi-Com*) is used for multi-communications among nodes. The communication between the node labelled ℓ and the node ℓ' succeeds, provided that (i) ℓ' belongs to the set L of possible receivers, (ii) the two nodes are compatible according to the compatibility function *Comp*, and (iii) that the first j values match with the evaluations of the first j terms in the input. When these comparisons are successful, the first node spawns a new process, running in parallel with the continuation P , whose task is to offer the output tuple to all its receivers L , except for ℓ' , which is removed, while in the second node each E_i is bound to each x_i . Outputs terminate when all the receivers in L have received the message (see the congruence rule). The transition label $\langle \ell_1 \text{ out}, \ell_2 \text{ in} \rangle$ records the fact that an inter-node communication occurred between the nodes ℓ_1 and ℓ_2 (with $\ell_2 \in L$).

The role of the compatibility function *Comp* is crucial in modelling real world constraints on communication. A basic requirement is that inter-node communications are mainly proximity-based, i.e. that only nodes

(Sensor-Com)

$$\frac{}{\ell : [\langle i, v_i \rangle . S_i \parallel S \parallel (i; z_i) . P \mid Q] \xrightarrow{\langle \ell \text{ out}, \ell \text{ in} \rangle} \ell : [S_i \parallel S \parallel P\{v_i/z_i\} \mid Q\{E_i/z_i\}]}$$

(Crypto-Sensor-Com)

$$\frac{}{\ell : [\langle \{i, v_i\}_K \rangle . S_i \parallel S \parallel (\{i; z_i\}_K) . P \mid Q] \xrightarrow{\langle \ell \{out\}, \ell \{in\} \rangle} \ell : [S_i \parallel S \parallel P\{v_i/z_i\} \mid Q\{E_i/z_i\}]}$$

(Intra-Com)

$$\frac{\bigwedge_{i=1}^k v_i = \llbracket E_i \rrbracket \wedge \bigwedge_{i=1}^j \llbracket E_i \rrbracket = \llbracket E'_i \rrbracket}{\ell : [\langle E_1, \dots, E_k \rangle . P \mid (E'_1, \dots, E'_j; x_{j+1}, \dots, x_k) . Q \parallel S]} \xrightarrow{\langle \ell \text{ out}, \ell \text{ in} \rangle}$$

$$\ell : [P \mid Q\{v_{j+1}/x_{j+1}, \dots, v_k/x_k\} \parallel S]$$

(Point2Point-Com)

$$\frac{Comp(\ell_1, \ell_2) \wedge \bigwedge_{i=1}^k v_i = \llbracket E_i \rrbracket \wedge \bigwedge_{i=1}^j \llbracket E_i \rrbracket = \llbracket E'_i \rrbracket}{\ell_1 : [\langle E_1, \dots, E_k \rangle . P_{11} \mid P_{12} \parallel S_P] \mid \ell_2 : [(E'_1, \dots, E'_j; x_{j+1}, \dots, x_k) . Q_{11} \mid Q_{12} \parallel S_Q]} \xrightarrow{\langle \ell_1 \text{ out}, \ell_2 \text{ in} \rangle}$$

$$\ell_1 : [\langle E_1, \dots, E_k \rangle . P_{11} \mid P_{12} \parallel S_P] \mid \ell_2 : [Q_{11}\{v_{j+1}/x_{j+1}, \dots, v_k/x_k\} \mid Q_{12} \parallel S_Q]$$

(Multi-Com)

$$\frac{\ell_2 \in L \wedge Comp(\ell_1, \ell_2) \wedge \bigwedge_{i=1}^k v_i = \llbracket E_i \rrbracket \wedge \bigwedge_{i=1}^j \llbracket E_i \rrbracket = \llbracket E'_i \rrbracket}{\ell_1 : [\langle \langle E_1, \dots, E_k \rangle \triangleright L . P_{11} \mid P_{12} \parallel S_P \rangle \mid \ell_2 : [(E'_1, \dots, E'_j; x_{j+1}, \dots, x_k) . Q_{11} \mid Q_{12} \parallel S_Q]]} \xrightarrow{\langle \ell_1 \text{ out}, \ell_2 \text{ in} \rangle}$$

$$\ell_1 : [\langle \langle E_1, \dots, E_k \rangle \triangleright L \setminus \{\ell_2\} . P_{11} \mid P_{12} \parallel S_P \rangle \mid \ell_2 : [Q_{11}\{v_{j+1}/x_{j+1}, \dots, v_k/x_k\} \mid Q_{12} \parallel S_Q]]$$

Table 1. Operational semantic rules for communication.

that are in the same transmission range can directly exchange messages. This is easily encoded here by defining a predicate (over node labels) yielding true only when two nodes are in the same transmission range. Of course, this function could be enriched in order to consider finer notions of compatibility expressing various policies, e.g. topics for event notification.

Hereafter, we assume the standard notion of transition system. Intuitively, a transition system is a graph, in which systems of nodes form the nodes and arcs represent the possible transitions between them (in our cases arcs come with labels). For technical reasons, which will be clear in the next section, hereafter, we will restrict ourselves to *finite* state systems, i.e. whose corresponding transition systems have a finite set of states. Note that this does not mean that the behaviour of such processes is finite, because their transition systems may have loops.

Example (cont'd) Consider our simple running example and the single run of the first system (the one of the second system is similar) where the four sensors of node ℓ_1 send a message to their control process P_c and P_c checks the collected data and sends the checking result to the node with label ℓ_2 . For brevity, we ignore their internal actions τ . We denote with P'_c (Q'_c , R'_c , resp.) the continuations of P_c (Q_c , R_c , resp.) after the first input prefixes, with v_{comp} the value $cmp(v_0, \dots, v_3)$, with v_{avg} the value $avg(v_0, \dots, v_3)$, and with v_{res_i} (with $i = 0, 1$) the value **ok** (**alarm** respectively), depending on which branch of the summation is chosen.

$$\begin{aligned}
N &= \ell_1 : [(0; z_0). P'_c \mid P \parallel (\langle 0, sense_0() \rangle. \tau. S_0 \parallel S_1 \parallel S_2 \parallel S_3)] \mid N_2 \mid N_3 \\
&\xrightarrow{\theta_0} \\
N' &= \ell_1 : [P'_c\{0/z_0\} \parallel (\tau. S_0 \parallel S_1 \parallel S_2 \parallel S_3)] \mid N_2 \mid N_3 \\
&\xrightarrow{\theta_1} \xrightarrow{\theta_2} \xrightarrow{\theta_3} \\
N'''' &= \ell_1 : [P'_c\{0/z_0, 1/z_1, 2/z_2, 3/z_3\} \parallel \\
&\quad (S_0 \parallel S_1 \parallel S_2 \parallel S_3)] \mid N_2 \mid N_3 \\
&= \\
&\quad \ell_1 : [\langle v_{comp}, v_{avg} \rangle \triangleright \{\ell_2\}. \tau. P_c \parallel (S_0 \parallel S_1 \parallel S_2 \parallel S_3)] \mid \\
&\quad \ell_2 : [\langle \mathbf{true}; x_{avg} \rangle. \langle \mathbf{ok}, x_{avg} \rangle \triangleright \{\ell_3\}. \tau. Q_c + \\
&\quad \quad \langle \mathbf{false}; x_{avg} \rangle. \langle \mathbf{alarm}, x_{avg} \rangle \triangleright \{\ell_3\}. \tau. Q_c] \mid N_3 \\
&\xrightarrow{\theta_{4i}} \\
N_i'''' &= \ell_1 : [P_c \mid P \parallel (S_0 \parallel S_1 \parallel S_2 \parallel S_3)] \mid \\
&\quad \ell_2 : [Q'_c\{v_{avg}/x_{avg}\} \parallel 0] \mid \\
&\quad \ell_3 : [(\tau. w_{res}, w_{avg}). \tau. R_c \parallel 0] \\
&= \\
&\quad \ell_1 : [P_c \mid P \parallel (S_0 \parallel S_1 \parallel S_2 \parallel S_3)] \mid \\
&\quad \ell_2 : [\langle v_{res_i}, v_{avg} \rangle \triangleright \{\ell_3\}. \tau. Q_c \parallel 0] \mid \\
&\quad \ell_3 : [(\tau. w_{res}, w_{avg}). \tau. R_c \parallel 0] \\
&\xrightarrow{\theta_{5i}} \\
N &= \ell_1 : [P_c \mid P \parallel (S_0 \parallel S_1 \parallel S_2 \parallel S_3)] \mid \\
&\quad \ell_2 : [Q_c \parallel 0] \mid \\
&\quad \ell_3 : [R'_c\{v_{res_i}/w_{res}, v_{avg}/w_{avg}\} \parallel 0] \\
&= \\
&\quad N_1 \mid N_2 \mid N_3
\end{aligned}$$

The whole sequence of transitions with source N is as follows.

$$N \xrightarrow{\theta_0} N' \xrightarrow{\theta_1} N'' \xrightarrow{\theta_2} N''' \xrightarrow{\theta_3} N'''' \xrightarrow{\theta_{4i}} \begin{cases} N_0'''' \xrightarrow{\theta_{50}} N \text{ if } i = 0 \\ N_1'''' \xrightarrow{\theta_{51}} N \text{ if } i = 1 \end{cases}$$

where N' , N'' , N''' , N'''' , N''''' represent the derivatives of N (i.e. the node systems reached in the computation) and the label θ_j , which denotes the

label of the j^{th} transition (θ_{ji} depending on the branch of the summation), are as follows.

$$\begin{aligned}\theta_0 &= \theta_2 = \langle \ell_1(j, v_j), \ell_1(j; z_i) \rangle, \\ \theta_1 &= \theta_3 = \langle \ell_1(\{j, v_j\}_{K_i}), \ell_1(\{j; z_j\}_{K_i}) \rangle, \\ \theta_{4i} &= \langle \ell_1(\langle cmp(v_0, \dots, v_3), avg(v_0, \dots, v_3) \rangle), \ell_2(v_{bool}; x_{avg}) \rangle \\ \theta_{5i} &= \langle \ell_2(\langle v_{res_i}, v_{avg} \rangle), \ell_3(w_{res}, w_{avg}) \rangle\end{aligned}$$

The evolution of the second system \hat{N} is similar and its transition labels $\hat{\theta}_i$ are as follows.

$$\begin{aligned}\hat{\theta}_0 &= \hat{\theta}_2 = \hat{\theta}_3 = \langle \ell_1(j, v_j), \ell_1(j; z_i) \rangle, \\ \hat{\theta}_1 &= \langle \ell_1(\{j, v_j\}_{K_i}), \ell_1(\{j; z_j\}_{K_i}) \rangle, \\ \hat{\theta}_{4i} &= \langle \ell_1(\langle halfcmp(v_0, v_1), avg(v_0, \dots, v_3) \rangle), \ell_2(v_{bool}; x_{avg}) \rangle \\ \hat{\theta}_{5i} &= \langle \ell_2(\langle v_{res_i}, v_{avg} \rangle), \ell_3(w_{res}, w_{avg}) \rangle\end{aligned}$$

Note that in both cases, the transition systems loop, but they are *finite* as required.

3 Stochastic Semantics

We now show how to extract quantitative information from a transition system by transforming it in a Continuous Time Markov Chains (CTMC) (see [27] for a more detailed description of this process). First, we introduce functions that associate costs to single transitions, by inspecting their enhanced labels. This information is sufficient to extract the necessary quantitative information to obtain the Continuous Time Markov Chains (CTMC). In general, by “cost” we mean any measure that affects quantitative properties of transitions: here, we intend the time the system is likely to remain within a given transition. We specify the cost of a system in terms of the time overhead due to its primitives. The cost of (the component of) the transition depends on both the current action and on its context of executions, in our case, on the nodes involved. Intuitively, cost functions define exponential distributions of transitions. Starting from them it is possible to compute the rates at which a system evolves and therefore the corresponding CTMC. Finally, to evaluate the system performance we need to compute the (unique) stationary distribution of the CTMC and the transition rewards.

3.1 Cost Functions

First, we intuitively present the main factors that influence the costs of actions and those due to their context. For the sake of simplicity, here

we ignore the costs for other primitives, e.g. restriction, constant invocation, parallel composition, summation, and internal actions (see [27] for a complete treatment).

- The cost of a *communication* depends on the costs of the input and output components. In particular, the cost of an (i) *output* depends on the size of the message and on the cost of each term of the message sent, in particular on its encryptions; (ii) *input* depends on the size of the message and on the cost of checks needed to accept the message. Actually, the two partners independently perform some low-level operations locally to their environment, each of which leads to a delay. Since communication is synchronous and handshaking, the overall cost corresponds to the cost paid by the slower partner.
- The cost of both *encryption* and *decryption* depends on the sizes of the cleartext and ciphertext, respectively, the complexity of the algorithm that implements it, the cipher mode adopted, and the kind of the key (short/long, short-term/long-term). The length of the key is important: usually, the longer the key, the greater the computing time. In addition, the cost for *decryption* depends on the cost of the checks needed to accept the decryption.

To define a cost function, we start by considering the execution of each action on a dedicated architecture that only has to perform that action, and we estimate the corresponding duration with a fixed rate r . Then we model the performance degradation due to the run-time support. To do that, we introduce a scaling factor for r in correspondence with each routine called by the implementation of the transition θ under consideration. Here, we just propose a format for these functions, with parameters to be instantiated on need. Note that these parameters depend on the node, e.g. in a node where the cryptographic operations are performed at very high speed (e.g. by a cryptographic accelerator), but with a slow link (low bandwidth), the time will be low for encryptions and high for communication; vice versa, in a node offering a high bandwidth, but poor cryptography resources.

Technically, we interpret costs as parameters of exponential distributions $F(t) = 1 - e^{-rt}$, with rate r and t as time parameter (general distributions are also possible see [29]). The *rate* r associated with the transition is the parameter that identifies the exponential distribution of the duration times of the transition, as usual in stochastic process algebras (e.g. [18,17]). The shape of $F(t)$ is a curve that grows from 0 asymptotically approaching 1 for positive values of its argument t . The

parameter r determines the slope of the curve: the greater r , the faster $F(t)$ approaches its asymptotic value. The probability of performing an action with parameter r within time x is $F(x) = 1 - e^{-rx}$, so r determines the time, Δt , needed to have a probability near to 1. The exponential distributions that we use enjoy the *memoryless property*, i.e. the occurrence of a new transition does not depend on when the previous transitions occurred. We also assume that transitions are *time homogeneous*, i.e. the corresponding rates do not depend on the time in which the transitions are fired.

We define in a few steps the function that associates rates with communication and decryption transitions, or, more precisely, with their enhanced labels. For simplicity, we assume the sensor cost of sensing from the environment as non-significant. We first give the auxiliary function $f_E : \mathcal{E} \rightarrow \mathbb{R}^+$ that estimates the effort needed to manipulate terms $E \in \mathcal{E}$.

- $f_E(a) = \text{size}(a)$
- $f_E(\{E_1, \dots, E_k\}_{E_0}) = f_{enc}(f_E(E_1), \dots, f_E(E_k), \text{kind}(E_0))$

The size of a name a ($\text{size}(a)$) matters. For an encrypted term, we use the function f_{enc} , which in turn depends on the estimate of the terms to encrypt and on the kind of the key (represented by $\text{kind}(E_0)$), i.e. on its length and on the corresponding crypto-system. Then we assign costs to communication and decryption actions.

- $\$_\alpha(\langle E_1, \dots, E_k \rangle) = f_{out}(f_E(E_1), \dots, f_E(E_k), bw)$
- $\$_\alpha(\langle E_1, \dots, E_j; x_{j+1}, \dots, x_k \rangle) = f_{in}(f_E(E_1), \dots, f_E(E_j), \text{match}(j), bw)$
- $\$_\alpha(\text{decrypt } E \text{ as } \{E_1, \dots, E_j; x_{j+1}, \dots, x_k\}_{E_0}) = f_{dec}(f_E(E), \text{kind}(E_0), \text{match}(j))$

The functions f_{out} and f_{in} define the costs of the routines that implement the send and receive primitives. Besides the implementation cost due to their own algorithms, the functions above depend on the bandwidth of the communication channel (represented by bw) and the cost of the exchanged terms, which is computed by the auxiliary function f_E . They in turn depend on the nodes where the communication occurs. Moreover, the inter-node communication depends on the proximity-relationship between the nodes, represented here by the function $F(\ell_O, \ell_I)$. Also, the cost of an input depends on the number of tests or matchings required (represented by $\text{match}(j)$). Finally, the function f_{dec} represents the cost of a decryption. It depends on the manipulated terms ($f_E(E)$), on the kind of key ($\text{kind}(E_0)$), on the number of matchings ($\text{match}(j)$), and on the cryptographic features of the node that performs the decryption.

Finally, the function $\$: \Theta \rightarrow \mathbb{R}^+$ associates rates with enhanced labels.

- $\$(\langle \ell_O \text{ out}, \ell_I \text{ in} \rangle) = F(\ell_O, \ell_I) \cdot \min\{\$_\alpha(\text{out}, \ell_O), \$_\alpha(\text{in}, \ell_I)\}$
- $\$(\ell \text{ dec}) = \$_\alpha(\text{dec}, \ell)$

As mentioned above, the two partners independently perform some low-level operations locally to their nodes, represented by the two node labels ℓ_O and ℓ_I . Each label leads to a delay in the rate of the corresponding action. Thus, the cost of the slower partner corresponds to the minimum cost of the operations performed by the participants, in isolation. Indeed the lower the cost, i.e. the rate, the greater the time needed to complete an action and hence the slower the speed of the transition occurring. The smaller r , the slower $F(t) = 1 - e^{-rt}$ approaches its asymptotic value.

Note that we do not fix the actual cost function: we only propose for it a set of parameters to reflect some features of an idealised architecture. Although very abstract, this suffices to make our point. A precise instantiation comes with the refinement steps from specification to implementations as soon as actual parameters become available.

Example (cont'd) We now associate a rate to each transition in the transition system of the system of nodes N , called N for brevity. To illustrate our methodology, we make some simplifying assumptions: we assume that τ transitions have no cost and that the coefficients due to the nodes amount to 1. We instantiate the cost functions given above, by using the following parameters each used to compute the rate corresponding to a particular action (sending, receiving and decryption) or a part of it, such as an encryption or a pattern matching: (i) **e** and **d** for encrypting and for decrypting; (ii) **s** and **r** for sending and for receiving; (iii) **m** for pattern matching; and (iv) **f** for the application of the aggregate function f , whose cost is proportional to the number of their arguments. The functions are:

- $f_E(a) = 1$
- $f_E(\{E_1, \dots, E_k\}_{E_0}) = \frac{\mathbf{e}}{\mathbf{s}} \cdot \sum_{i=1}^k f_E(E_i) + 1$
- $\$_\alpha(\langle E_1, \dots, E_k \rangle) = \frac{1}{\mathbf{s} \cdot \sum_{i=1}^k f_E(E_i)}$
- $\$_\alpha((E_1, \dots, E_j; x_{j+1}, \dots, x_k)) = \frac{1}{\mathbf{r} \cdot k + \mathbf{m} \cdot j}$
- $\$_\alpha(\text{decrypt } E \text{ as } \{E_1, \dots, E_j; x_{j+1}, \dots, x_k\}_{E_0}) = \frac{1}{\mathbf{d} \cdot k + \mathbf{m} \cdot j}$
- $\$_\alpha(f(E_1, \dots, E_k)) = \frac{1}{\mathbf{f} \cdot k}$

Intuitively, these parameters represent the time spent performing the corresponding action on a single term. They occur in the denominator, therefore keeping the rule that the faster the time, the slower the rate. Since

transmission is usually more time-consuming than the corresponding reception, the rate of a communication, will always be that of output. The rates of the transitions of N and \hat{N} are $c_j = \$(\theta_j)$ and $\hat{c}_j = \$(\hat{\theta}_j)$, with $j \in [0, 5]$ and $i \in [0, 1]$.

$$\begin{aligned} c_0 &= c_2 = \frac{1}{2s}, & \hat{c}_0 &= \hat{c}_2 = \hat{c}_3 = \frac{1}{2s}, \\ c_1 &= c_3 = \frac{1}{2e+s}, & \hat{c}_1 &= \frac{1}{2e+s} \\ c_{4i} &= \frac{1}{8f+2s} & \hat{c}_{4i} &= \frac{1}{6f+2s} \\ c_{5i} &= \frac{1}{s} & \hat{c}_{5i} &= \frac{1}{s} \end{aligned}$$

For instance, the rate c_1 of the second transition is: $c_1 = \$(\theta_1) = \frac{1}{2e+s}$, where $\frac{1}{2e+s} = \min\{\frac{1}{2e+s}, \frac{1}{2d+r+m}\}$. Note that our costs can be further refined; we could e.g. use a different rate for transmission when internal to a node (costs c_j and \hat{c}_j with $j \in [0, 3]$) and when external (costs c_{ji} and \hat{c}_{ji} with $j \in [3, 4]$).

3.2 Stochastic Analysis

Now, we derive a Continuous Time Markov Chain (CTMC) from a transition system. Afterwards, we can calculate the actual performance measures, e.g. the throughput or utilisation of a certain resource (see [2,26] for more details on the theory of stochastic processes).

Markov Chains We use the rates of transitions computed in Subsect. 3.1, to transform a transition system N into its corresponding $CTMC(N)$.

Actually, the *transition rate* $q(N_i, N_j)$ at which a system changes from behaving like process N_i to behaving like N_j is the sum of the single rates ϑ_k of all the possible transitions from N_i to N_j . Given a transition system N , the corresponding CTMC has a state for each node in N , and the arcs between states are obtained by coalescing all the arcs with the same source and target in N . Recall that a CTMC can be seen as a directed graph and that its matrix \mathbf{Q} , the *generator matrix*, (apart from its diagonal) represents its adjacency matrix. Note that $q(N_i, N_j)$ coincides with the off-diagonal element q_{ij} of the generator matrix \mathbf{Q} . Hence, hereafter we will use indistinguishably CTMC and its corresponding \mathbf{Q} to denote a Markov chain. More formally, the entries of \mathbf{Q} are defined as follows. Given a transition system N , the corresponding CTMC has a state for each node in N , and the arcs between states are obtained by coalescing all the arcs with the same source and target in N . Recall that a CTMC can be seen as a directed graph and that its matrix \mathbf{Q} , called *generator matrix*, (apart from its diagonal) represents its adjacency matrix. Note

that $q(N_i, N_j)$ coincides with the off-diagonal element q_{ij} of the generator matrix \mathbf{Q} . Hence, hereafter we will use indistinguishably CTMC and its corresponding \mathbf{Q} to denote a Markov chain. More formally, the entries of \mathbf{Q} are defined as follows.

$$q_{ij} = \begin{cases} q(N_i, N_j) = \sum_{N_i \xrightarrow{\theta_k} N_j} \$(\theta_k) & \text{if } i \neq j \\ -\sum_{j=0, j \neq i}^n q_{ij} & \text{if } i = j \end{cases}$$

Evaluating the Performance Performance measures should be taken over long periods of time to be significant. These measures are usually obtained by resorting to stationary probability distributions of CTMCs. The *stationary probability distribution* of a CTMC is $\Pi = (X_0, \dots, X_{n-1})$ such that Π solves the matrix equation $\Pi^T \mathbf{Q} = \mathbf{0}$ and $\sum_{i=0}^n X_i = 1$. If the transition system is finite and has a cyclic initial state, then there exists a unique stationary probability distribution.

Example (cont'd) Consider the transition system corresponding to the system of nodes N that is, as required above, finite and with a cyclic initial state. We derive the following generator matrix \mathbf{Q}_1 of $CTMC(N)$ and the corresponding stationary distribution is Π_1 , where $C = 4s + 2e + 2f$, by solving the system of linear equations $\Pi_1^T \mathbf{Q}_1 = \mathbf{0}$ and $\sum_{i=0}^n X_i = 1$, where $\Pi_1 = (X_0, \dots, X_6)$. Similarly, we can derive the generator matrix \hat{Q}'_1 and the corresponding stationary distribution $\hat{\Pi}_1$ for the transition system corresponding to \hat{N} , where $\hat{C} = 9s + 2e + 3f$.

$$\mathbf{Q}_1 = \begin{bmatrix} -c_0 & c_0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -c_1 & c_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -c_2 & c_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -c_3 & c_3 & 0 & 0 \\ 0 & 0 & 0 & 0 & -(c_{40} + c_{41}) & c_{40} & c_{41} \\ c_{50} & 0 & 0 & 0 & 0 & -c_{50} & 0 \\ c_{51} & 0 & 0 & 0 & 0 & 0 & -c_{51} \end{bmatrix}$$

$$\Pi_1 = \left[\frac{s}{C}, \frac{(2e + s)}{2C}, \frac{s}{2C}, \frac{(2e + s)}{C}, \frac{(4f + s)}{2C}, \frac{s}{4C}, \frac{s}{4C} \right]$$

$$\hat{\Pi}_1 = \left[\frac{2s}{\hat{C}}, \frac{(2e + s)}{\hat{C}}, \frac{2s}{\hat{C}}, \frac{2s}{\hat{C}}, \frac{(3f + s)}{\hat{C}}, \frac{s}{2\hat{C}}, \frac{s}{2\hat{C}} \right]$$

To define performance measures for a system N , we define a *reward structure* associated with N , following [19,18,12]. Usually, a reward structure is simply a function that associates a reward with any state passed

through in a computation of N . For instance, when calculating the utilisation of a resource, we assign value 1 to any state in which the use of the resource is enabled (typically the source of a transition that uses the resource). All the other states earn the value 0. We use a slightly different notion, where rewards are computed from rates of transitions [27]. To measure instead the throughput of a system, i.e. the amount of useful work accomplished per unit time, a reasonable choice is to use as nonzero reward a value equal to the rate of the corresponding transition. The reward structure of a system N is a vector of rewards with as many elements as the number of states of N . By looking at the probability stationary distribution of and varying the reward structure, we can compute different performance measures. The *total reward* is obtained by summing the values of the stationary distribution Π multiplied by the corresponding reward structure ρ .

Definition 2. *Given a system N , let $\Pi = (X_0, \dots, X_{n-1})$ be its stationary distribution and $\rho = \rho(0), \dots, \rho(n-1)$ be its reward structure. The total reward of N is computed as $R(N) = \sum_i \rho(i) \cdot X_i$.*

Example (cont'd) To evaluate the relative efficiency of the two systems of nodes, we compare the throughput of both, i.e. the number of instructions executed per time unit. The throughput for a given activity is found by first associating a transition reward equal to the activity rate with each transition. In our systems each transition is fired only once. Also, the graph of the corresponding CTMC is cyclic and all the labels represent different activities. This amounts to saying that the throughput of all the activities is the same, and we can freely choose one of them to compute the throughput of N . Thus we associate a transition reward equal to its rate with the last communication and a null transition reward with all the others communications. The total reward $R(N)$ of the system amounts then to $\frac{1}{2(8s+4e+4f)}$, while $R(\hat{N})$ amounts to $\frac{1}{2(9s+2e+3f)}$. By comparing the two throughputs, it is straightforward to obtain that $R(N) < R(\hat{N})$, i.e. that, as expected, \hat{N} perform better. To use this measure, it is necessary to instantiate our parameters under various hypotheses, depending on several factors, such as the network load, the packet size, and so on. Furthermore, we need to consider the costs of cryptographic algorithms and how changing their parameters impact on energy consumption and on the guaranteed security level (see e.g. [24]).

4 Conclusions

In the IoT setting the risk for devices of being attacked is higher and higher, and still security is not taken sufficiently into account, since supporting security in an affordable way is quite challenging. We have presented the first steps towards a framework and formal design methodology that support designers in specifying an IoT system and in estimating the cost of security mechanisms starting from its specification. In this way, it suffices to have information about the activities performed by the components of a system in isolation, and about some features of the network architecture. A key feature of our approach is that quantitative aspects are symbolically represented by parameters. Actual values are obtained as soon as the designer provides some additional information about the hardware architecture and the cryptographic algorithms relative to the system in hand. By abstractly reasoning about these parameters designers can compare different implementations of the same IoT system, and choose among different alternatives the one that ensures the best trade-off between security guarantees and their price.

In practice, we proposed the process algebra IoT-LYSA, an extension of LYSA with suitable primitives for describing the activity of sensors and of sensor nodes, and for describing the possible patterns of communication among the IoT entities. We have equipped the calculus with an enhanced semantics, following [15], where each system transition is associated to a rate in the style of [27]. Starting from the information about the rates of system activities, it is possible to mechanically derive Markov chains through which we can perform cost evaluation by using standard techniques and tools [33,30,32].

Our approach follows the well-established line of research about performance evaluation through process calculi and probabilistic model checking (see [13,20,21] for a survey). To the best of our knowledge, the application of formal methods to IoT systems or to wireless or sensor networks have not been largely studied and only a limited number of papers in the literature addressed the problem from a process algebras perspective, e.g. [22,23,10,31] to cite only a few. In [11] the problem of modelling and estimating the communication cost in an IoT scenario is tackled through Stochastic Petri Net. Their approach is similar to ours: they derive a CTMC from a Petri Net describing the system and proceed with the performance evaluation by using standard tools. Differently from us, they focus not on the cost of security but only on the one of communication (they do not use cryptographic primitives). In [16] a performance compar-

ison between the security protocols IPSec and DTLS is presented, in particular by considering impact on the resources of IoT devices with limited computational capabilities. They modified protocols implementations to make them properly run on the devices. An extensive experimental evaluation study on these protocols shows that both their implementations ensure a good level of end-to-end security.

References

1. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols - The Spi calculus. *Information and Computation*, 148(1):1–70, 1999.
2. A. A. Allen. *Probability, Statistics and Queueing Theory with Computer Science Applications*. Academic Press, 1978.
3. M. Andreessen. *Why Software Is Eating The World*. The Wall Street Journal, August 20, 2011.
4. C. Bodei, L. Brodo, R. Focardi. Static Evidences for Attack Reconstruction. *Programming Languages with Applications to Biology and Security 2015*. LNCS 9465, pp.162-182, Springer, 2015.
5. C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Automatic validation of protocol narration. *Proc. of CSFW'03*, pages 126–140. IEEE, 2003.
6. C. Bodei, M. Buchholtz, M. Curti, P. Degano, F. Nielson, and H. Riis Nielson, C. Priami. On Evaluating the Performance of Security Protocols *Proc. of PaCT'05*, LNCS 3606, pp. 115, 2005
7. C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Static validation of security protocols. *Journal of Computer Security* 13(3): 347-390 (2005)
8. M. Bravetti, M. Bernardo and R. Gorrieri. Towards Performance Evaluation with General Distributions in Process Algebras. *Proc. of CONCUR'98*, LNCS 1466, 1998.
9. M. Buchholtz, F. Nielson, and H. Riis Nielson. A calculus for control flow analysis of security protocols. *International Journal of Information Security*, 2 (3-4), 2004.
10. V. Castiglioni, R. Lanotte and M. Merro. *A Semantic Theory for the Internet of Things*. arXiv:1510.04854v1.
11. L. Chen, L. Shi, W. Tan. *Modeling and Performance Evaluation of Internet of Things based on Petri Nets and Behavior Expression*. Research Journal of Applied Sciences, Engineering and Technology 4(18): 3381-3385, 2012.
12. G. Clark. Formalising the specifications of rewards with PEPA. *Proc. of PAPM'96*, pp. 136-160. CLUT, Torino, 1996.
13. A. Clark, S. Gilmore, J. Hillston and M. Tribastone. *Stochastic Process Algebras*. Formal Methods for the Design of Computer, Communication, and Software Systems (SFM), 2007.
14. P. Degano and C. Priami. Non Interleaving Semantics for Mobile Processes. *Theoretical Computer Science*, 216:237–270, 1999.
15. P. Degano and C. Priami. Enhanced Operational Semantics. *ACM Computing Surveys*, 33, 2 (June 2001), 135-176.
16. A. De Rubertis, L. Mainetti, V. Mighali, L. Patrono, I. Sergi, M.L. Stefanizzi, S. Pascali, *Performance evaluation of end-to-end security protocols in an Internet of Things* in 21st International Conference on Software, Telecommunications and Computer Networks (SoftCOM), 2013.

17. H. Hermanns and U. Herzog and V. Mertsiotakis. Stochastic process algebras – between LOTOS and Markov Chains. *Computer Networks and ISDN systems* 30(9-10):901-924, 1998.
18. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
19. R. Howard. *Dynamic Probabilistic Systems: Semi-Markov and Decision Systems*. Volume II, Wiley, 1971.
20. M. Kwiatkowska, G. Norman and D. Parker. *Stochastic Model Checking*. Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07), LNCS 4486, 2007
21. M. Kwiatkowska and D. Parker. Advances in Probabilistic Model Checking. *Software Safety and Security - Tools for Analysis and Verification*: 33:126-151, IOS Press, 2012.
22. I. Lanese and D. Sangiorgi. An operational semantics for a calculus for wireless systems. *Theoretical Computer Science* 411(19): 1028-1948 (2010)
23. I. Lanese, L. Bedogni and M.D. Felice. *Internet of Things: A Process Calculus Approach*. Proc. of the 28th Annual ACM Symposium on Applied Computing (ACM SAC '13), 2013
24. J. Lee, K. Kapitanova, S.H. Son: The price of security in wireless sensor networks. *Computer Networks* 54(17): 2967-2978 (2010)
25. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (I and II). *Information & Computation*, 100(1):1–77, 1992.
26. R. Nelson. *Probability, Stochastic Processes and Queuing Theory*. Springer, 1995.
27. C. Nottegar, C. Priami and P. Degano. Performance Evaluation of Mobile Processes via Abstract Machines. *Transactions on Software Engineering*, 27(10), 2001.
28. G. Plotkin. A Structural Approach to Operational Semantics. *Tech. Rep. Aarhus University, Denmark*, 1981, DAIMI FN-19
29. C. Priami. Language-based Performance Prediction of Distributed and Mobile Systems *Information and Computation* 175: 119-145, 2002.
30. A. Reibnam and R. Smith and K. Trivedi. Markov and Markov reward model transient analysis: an overview of numerical approaches. *European Journal of Operations Research*: 40:257-267, 1989.
31. A. Singh, C.R. Ramakrishnan and S.A. Smolka. A process calculus for Mobile Ad Hoc Networks. *Science of Computer Programming* 75(6): 440-469 (2010)
32. W. J. Stewart. *Introduction to the numerical solutions of Markov chains*. Princeton University Press, 1994.
33. K. S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. Edgewood Cliffs, NY, 1982.